Soporte de sistema operativo para ahorro de energía en plataformas móviles con procesadores multicore asimétricos

TRABAJO FIN DE GRADO

Adrián García García
Álvaro Sanz del Río

Dirigido por: Juan Carlos Sáez Alcaide

Grado en Ingeniería Informática /
Grado en Ingeniería de Computadores
Facultad de Informática
Universidad Complutense de Madrid

Junio 2016

Soporte de sistema operativo para ahorro de energía en plataformas móviles con procesadores multicore asimétricos

Memoria de Trabajo Fin de Grado

Adrián García García
Álvaro Sanz del Río

Dirigido por: Juan Carlos Sáez Alcaide

Grado en Ingeniería Informática /
Grado en Ingeniería de Computadores
Facultad de Informática
Universidad Complutense de Madrid

Junio 2016

Documento maquetado con TEXIS v.1.0.

Este documento está preparado para ser imprimido a doble cara.

Autorización de difusión y utilización

Los abajo firmantes, alumno/s y tutor/es del Trabajo Fin de Grado (TFG) en el Grado en Ingeniería Informática y Grado en Ingeniería de Computadores de la Facultad de Informática, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente el Trabajo Fin de Grado (TFG) cuyos datos se detallan a continuación. Así mismo autorizan a la Universidad Complutense de Madrid a que sea depositado en acceso abierto en el repositorio institucional con el objeto de incrementar la difusión, uso e impacto del TFG en Internet y garantizar su preservación y acceso a largo plazo.

Título: Soporte de sistema operativo para ahorro de energía en plataformas móviles con procesadores multicore asimétricos

Curso académico: 2015/2016

Alumnos: Adrián García García y Álvaro Sanz del Río

Director: Juan Carlos Sáez Alcaide, Departamento de Arquitectura de Computadores y Automática

Firma del alumno/s	Firma del tutor/es

Madrid, a 14 de junio de 2016.

"Lo que sabemos es una gota de agua,
lo que ignoramos un océano."
– Isaac Newton

"The isolated man does not develop any intellectual power. It is necessary for him to be immersed in an environment of other men, whose techniques he absorbs during the first twenty years of his life. He may then perhaps do a little research of his own and make a very few discoveries which are passed on to other men. From this point of view the search for new techniques must be regarded as carried out by the human community as a whole, rather than by individuals."
– Alan Turing

Agradecimientos

Nuestro más sincero agradecimiento a Juan Carlos por la infinita paciencia y ayuda que nos ha prestado, creemos que ha sido la principal causa de que este complejo Trabajo de Fin de Grado llegase a buen puerto. También queremos agradecer a la Universidad Complutense de Madrid y a todos los alumnos y profesores cuyo trabajo previo nos ha permitido realizar este Trabajo de Fin de Grado.

Resumen

Los procesadores multicore asimétricos con repertorio común de instrucciones (AMPs-Asymmetric Multicore Processors) han sido propuestos recientemente como alternativa de bajo consumo a los procesadores multicore simétricos convencionales. Los AMPs combinan, en un mismo chip, cores rápidos de alto rendimiento, con cores más lentos y sencillos de consumo reducido. Uno de los ejemplos más destacados de procesador multicore asimétrico es el procesador big.LITTLE de ARM, que incorporan algunos modelos de teléfonos móviles y tablets disponibles en la actualidad.

Trabajos previos han demostrado que para explotar los beneficios potenciales de los procesadores multicore asimétricos, el sistema operativo debe tener en cuenta el beneficio relativo (*speedup*) que cada aplicación experimenta al ejecutar en un core rápido frente a un core lento. Actualmente, los planificadores por defecto de los sistemas operativos de propósito general no tienen en cuenta la diversidad de *speedups* entre aplicaciones que puede estar presente en una carga de trabajo multiprogramada. En consecuencia, la asignación de aplicaciones a cores que hacen estos planificadores no extrae el máximo rendimiento por vatio de la plataforma.

Recientemente se han realizado extensiones en el kernel Linux para ofrecer un mejor soporte de planificación en multicore asimétricos. Sin embargo, estas extensiones del planificador, utilizadas fundamentalmente en dispositivos móviles con el sistema operativo Android, tampoco tienen en cuenta la diversidad de *speedups* en las aplicaciones de la carga de trabajo. Por lo tanto estas extensiones no constituyen una aproximación robusta desde el punto de vista de la eficiencia energética.

En este proyecto se lleva a cabo la evaluación exhaustiva de distintos algoritmos de planificación para multicore asimétricos sobre una plataforma provista de un procesador ARM big.LITTLE. El principal objetivo del estudio es cuantificar el grado de eficiencia energética y el rendimiento global proporcionado por implementaciones de estos algoritmos en el kernel Linux sobre hardware multicore asimétrico real.

Palabras clave: Procesadores multicore asimétricos, Kernel Linux, Planificación, Eficiencia energética, Monitorización hardware.

Abstract

Asymmetric single-ISA (instruction set architecture) multicore processors (AMPs) have been recently proposed as a more power-efficient alternative to conventional symmetric multicore processors. AMPs combine high-performance big cores with power-efficient small cores on a single chip. The big.LITTLE ARM processor, which is integrated in many commercial off-the-shelf mobile devices, constitutes a remarkable example of asymmetric multicore processor.

In order to fully tap into the potential of AMPs, the operating system scheduler must factor in, when making scheduling decisions, the relative benefit (aka. *speedup*) that each application derives when running on a big core relative to a small one. Notably, current default schedulers in general-purpose operating systems do not take into account the diversity in relative speedups that may be present in a multi-program workload. As a result, these schedulers do not always guarantee the maximum performance-per-watt on asymmetric multicore platforms.

Recently, scheduling extensions have been created for the Linux kernel to offer a better support on asymmetric multicore systems. However, these scheduling extensions (used primarily on mobile devices running the Android operating system) do not take into consideration the relative speedup of each application when making scheduling decisions. Consequently, these extensions do not constitute a robust approach from the energy efficiency standpoint.

In this project, we carry out a comprehensive evaluation of different asymmetry-aware scheduling algorithms on a system featuring an ARM big.LITTLE processor. The main goal of the study is to assess the degree of energy efficiency and overall system performance of implementations of these algorithms in the Linux kernel running on real asymmetric hardware.

Keywords: Asymmetric Multicore, Linux kernel, Scheduling, Energy Efficiency, Hardware Monitoring.

Índice

Índice de figuras

Índice de Tablas

Capítulo 1

Introducción

La evolución tecnológica de los componentes electrónicos ha seguido un crecimiento de carácter exponencial en las últimas décadas. Esto ha propiciado una transición desde los gigantescos *mainframes*, a los potentes dispositivos móviles disponibles en la actualidad, que ponen a nuestra disposición todas las herramientas software imaginables en la palma de nuestra mano. Desde su publicación en 1965, la famosa Ley de Moore ha permanecido vigente como un mandamiento de la Informática. Sin embargo, el deber de su cumplimiento ha llevado a los fabricantes de procesadores a mantener un ritmo de renovación tecnológica basado en el diseño de procesadores que operen a mayor frecuencia de trabajo. Esto dejó de lado otras técnicas que mejoran el rendimiento que se están recuperando hoy en día, como la explotación del paralelismo, véase antiguas arquitecturas vectoriales como el Cray-1, ejemplo de arquitectura SIMD (*Single Instruction Multiple Data*).

El paradigma de diseño de microprocesadores ha evolucionado mucho en los últimos tiempos. La rápida escalada del consumo energético y problemas de disipación térmica pusieron en evidencia que el incremento de frecuencia como técnica de mejora de rendimiento no podía mantenerse. A partir de ese momento, se ha optado por incluir un mayor número de elementos de procesamiento (cores) en un mismo *chip*, lo cual ha sido posible gracias a la disminución del tamaño de los transistores. Estos nuevos diseños complementados con otras mejoras arquitectónicas permitieron realizar un mayor uso del paralelismo a nivel de hilo (*Thread Level Parallelism* - TLP), mediante la inclusión de múltiples unidades funcionales en un mismo core (p.ej., *simultaneous multithreading*).

Los procesadores multicore convencionales, que están formados por cores idénticos, están presentes en un gran número de dispositivos electrónicos. Destacaremos dos clases fundamentales:

- Los procesadores de alto rendimiento, que integran cores de elevado consumo y alta frecuencia de trabajo, como el Intel Core i7-6700K. Estos cores basan su diseño en la optimización de la capacidad de cómputo mediante el uso de técnicas como la ejecución fuera de orden o el lanzamiento múltiple, con un claro aumento del consumo energético como contrapartida.

- Los procesadores constituidos por cores de baja frecuencia y consumo reducido, cuya filosofía se centra en el ahorro energético, como los Intel Atom o los ARM Cortex A7 o A53. Estos procesadores están presentes en dispositivos móviles en los que el ahorro de batería es un aspecto fundamental. Cabe destacar que estos procesadores, pueden ofrecer buen rendimiento en aplicaciones con gran paralelismo a nivel de hilo (*Thread-Level Parallelism* - TLP), ya que las paradas de *pipeline* que pueden suceder mientras un hilo se ejecuta en un core no impiden que otros hilos prosigan su ejecución en otros cores.

Los ejemplos propuestos disponen de un número limitado de cores, comúnmente entre dos y cuatro. Pero estos tipos de procesador se extrapolan a necesidades de computación más elevadas. En el primer ámbito se encuentran los procesadores usados en servidores y supercomputación, por ejemplo el Intel Xeon E7-8870 v3, que cuenta con 18 cores. En la segunda clase tenemos un ejemplo mucho más extendido en todo tipo de dispositivos, y son las unidades de procesamiento gráfico (*Graphics Processor Unit* - GPU). Por ejemplo, la nueva tarjeta gráfica NVIDIA GTX 1080, contiene 2560 cores CUDA. A pesar de su diseño específico para procesamiento gráfico, su capacidad para realizar eficientemente operaciones sencillas en paralelo sobre cantidades masivas de datos, ha provocado que el uso de GPUs se extienda a otros campos como la minería de datos, HPC o la inteligencia artificial.

Cabe destacar que las aplicaciones actuales presentan necesidades de cómputo de muy diversa índole. Por una parte, hay aplicaciones que pasan una gran parte del tiempo realizando cálculos (*CPU intensive*) y además suelen explotar bien los principios de localidad espacial y temporal en los accesos a cache. Para estas aplicaciones, el rendimiento no se ve afectado sustancialmente por las características de la jerarquía de memoria. Por otro lado, existen aplicaciones (*memory intensive*) que provocan numerosas paradas en el *pipeline* del procesador esperando a que se resuelvan fallos de cache de alta latencia.

La existencia de aplicaciones con distintas necesidades de cómputo provocan que los procesadores multicore convencionales constituyan una alternativa subóptima en ciertos escenarios. Por ejemplo, ejecutar una tarea intensiva en memoria en un core de alto rendimiento no permite extraer todo el potencial de las características microarquitectónicas, destinadas a extraer el máximo paralelismo a nivel de instrucción. Por el contrario, la ejecución de un programa intensivo en CPU en un procesador de bajo consumo no constituye la mejor opción desde el punto de vista de la eficiencia energética, ya que a pesar del menor consumo de potencia del core, la energía consumida durante la ejecución puede ser elevada debido al pobre rendimiento que caracteriza a este tipo de cores.

La diversificación de las necesidades de computación que presentan las aplicaciones ha propiciado el inicio de una nueva etapa en el diseño de *hardware*, en la que se incluyen en un mismo *chip* componentes especializados en resolver cierto tipo de problemas. En este contexto se propusieron como alternativa los procesadores multicore asimétricos [16, 17] que proporcionan un mejor soporte que los multicore convencionales para cargas de trabajo heterogéneas. Este Trabajo Fin de Grado se centra en el problema de la planificación de procesos a nivel de sistema operativo sobre sistemas multicore asimétricos.

El resto de este capítulo se estructura de la siguiente forma. En la Sección 1.1 se introduce el concepto de procesador multicore asimétrico. La Sección 1.2 ilustra la motivación de este Trabajo Fin de Grado. En la Secciones 1.3 y 1.4 se presentan los objetivos del proyecto y el plan de trabajo para el mismo. Finalmente, la Sección 1.5 describe la estructura de la memoria.

1.1. Procesadores multicore asimétricos

Un procesador multicore asimétrico (*Asymmetric Multicore Processor* - AMP) integra cores con distintas características en un mismo chip. Por un lado, los AMPs integran un grupo de cores rápidos, de alto rendimiento y consumo, que trabajan a alta frecuencia e implementan complejas técnicas microarquitectónicas como la ejecución fuera de orden o el lanzamiento múltiple de instrucciones. Por otra parte, estos procesadores incluyen un grupo de cores más lentos y sencillos. Estos cores operan a una frecuencia de trabajo más baja e implementan un *pipeline* más sencillo (p.ej., con ejecución en orden) y de bajo consumo. Debido a su mayor complejidad, los cores de alto rendimiento típicamente ocupan un mayor porcentaje del área del chip que los cores de bajo consumo [17, 12]. A lo largo de la memoria nos referiremos a los cores de alto rendimiento como rápidos o big, y a los cores de bajo consumo, como lentos o small.

Los procesadores multicore asimétricos aportan una mayor flexibilidad que los procesadores convencionales, ya que mantienen la capacidad de un procesador de alto rendimiento para aplicaciones intensivas en CPU, sin sacrificar la eficiencia energética. Nótese que las aplicaciones intensivas en memoria, que provocan muchas paradas en el *pipeline*, se podrían ejecutar en un core lento sin penalizar su rendimiento de forma significativa, pero proporcionando una considerable mejora en la eficiencia energética del sistema.

El ejemplo más destacado a nivel comercial de multicore asimétrico es el procesador big.LITTLE de ARM [2], que está presente en múltiples dispositivos móviles actuales. Intel también ha mostrado su interés por este tipo de arquitecturas y el prototipo QuickIA[4] es un claro ejemplo de ello. En los procesadores multicore asimétricos actuales, los distintos cores exponen un repertorio de instrucciones común, lo cual facilita enormemente el desarrollo de software [25]. Esta aproximación contrasta con la que se sigue en otros sistemas heterogéneos como el IBM Cell BE [9], donde los distintos cores exponen un repertorio de instrucciones diferente.

La eficiencia energética resulta un aspecto clave en el entorno de computación de alto rendimiento (HPC), dónde uno de los mayores costes es la factura eléctrica. Las arquitecturas AMP podrían ser de gran utilidad en este ámbito, combinadas, por ejemplo con paradigmas de programación paralela como OpenMP. Esto permitiría a los programadores controlar la ejecución de determinadas tareas e incluso fases de aplicaciones en determinados cores, pudiendo explotar el paralelismo de forma más eficiente. Por ejemplo, usar los cores rápidos para ejecutar las fases secuenciales intensivas en CPU de los

procesos y los lentos y eficientes para ejecutar fases multihilo, puede proporcionar mayor eficiencia energética [1]. Por otra parte, es posible mejorar el rendimiento global si los cores rápidos se destinan a la ejecución de fases intensivas en CPU de un programa, y los lentos se destinan a la ejecución de las fases intensivas en memoria[17, 36].

1.2. Motivación

Trabajos previos han demostrado que para explotar el potencial de los procesadores multicore asimétricos, el sistema operativo debe tener en cuenta el beneficio relativo (*speedup*) que cada aplicación experimenta al ejecutar en un core rápido frente a un core lento [17, 15, 31]. Algunas aplicaciones explotan de forma eficiente las características microarquitectónicas presentes en los cores de alto rendimiento; estas aplicaciones habitualmente experimentan un *speedup* significativo en estos cores en comparación con la ejecución en un core lento. Por el contrario, otras aplicaciones, que provocan frecuentes paradas del *pipeline* del procesador, no ven mejorado su rendimiento de forma significativa al ejecutarse en un core de alto rendimiento. En las decisiones de planificación se debería tener en cuenta el *speedup* relativo de cada aplicación para optimizar tres aspectos esenciales: la productividad o rendimiento global, la eficiencia energética y la justicia. En el Capítulo 3 se presentan distintas métricas para cuantificar la efectividad de distintos algoritmos en esos tres aspectos.

En la actualidad los planificadores por defecto de los sistemas operativos de propósito general no explotan la diversidad de *speedups* en una carga de trabajo multiprogramada. Recientemente se han llevado a cabo extensiones en el kernel Linux para proporcionar mejor soporte de planificación para sistemas multicore asimétricos en entornos interactivos. En la actualidad, estas extensiones se utilizan principalmente en dispositivos móviles con el sistema operativo Android (basado en el kernel Linux). Cabe destacar que estas extensiones de planificación denominadas HMP (*Heterogeneous Multi-Processing*) no forman parte de la versión *mainline* del kernel. Los cambios pertinentes se distribuyen en el parche `SCHED_HMP`[24], que introduce la siguiente funcionalidad en el planificador. El sistema operativo clasifica las tareas (hilos de ejecución) en dos categorías: *light* y *heavy*. Las tareas en la categoría *light* son aquellas que pasan gran parte del tiempo bloqueadas por E/S (p.ej., interacción con el usuario). Por el contrario, una tarea *heavy* dedica la mayor parte de su tiempo de ejecución a realizar cómputos en la CPU. En este contexto, el planificador asigna de forma preferente las tareas *heavy* a cores rápidos de alto rendimiento, y relega las tareas *light* a cores lentos.

A pesar de que el parche `SCHED_HMP` mejora la eficiencia energética en entornos interactivos, las extensiones de planificación que introduce presentan dos limitaciones importantes. En primer lugar, a la hora de clasificar tareas en *light* y *heavy*, no se tiene en cuenta el beneficio relativo (*speedup*) que una tarea experimenta al ejecutar en un core de alto rendimiento frente a uno de bajo consumo. En particular, todas las tareas intensivas en cómputo se clasifican como *heavy*, con independencia del *speedup* que derivan al usar un core de alto rendimiento o de su grado de utilización de la jerarquía de memoria. Como consecuencia, las asignaciones de tareas a cores no son óptimas desde el punto

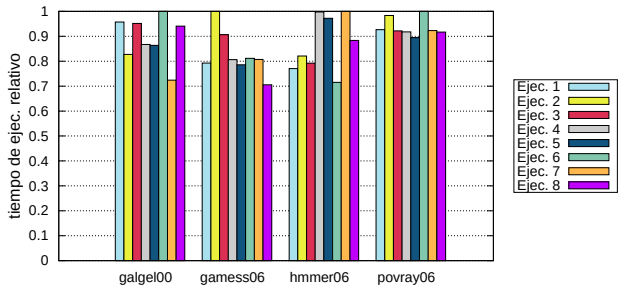

Figura 1.1: Tiempo de ejecución normalizado de las distintas aplicaciones de la carga para el planificador SCHED_HMP.

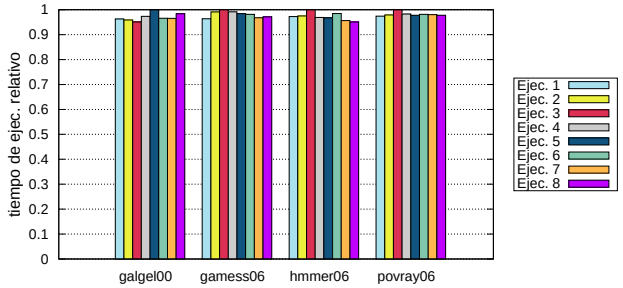

Figura 1.2: Tiempo de ejecución normalizado de las distintas aplicaciones de la carga para el planificador RR.

de vista de la eficiencia energética o el rendimiento global. En segundo lugar, cuando el número de tareas *heavy* supera el número de cores de alto rendimiento, algunas tareas de este tipo quedan relegadas a cores de bajo consumo. Además, el planificador no realiza ninguna acción para garantizar que todas las tareas de este tipo usen los cores rápidos de forma justa para que todas las tareas realicen un progreso similar en el sistema asimétrico. Esto constituye una importante limitación en el contexto de la computación de alto rendimiento, donde la mayor parte de aplicaciones de la carga son intensivas en cómputo (*heavy tasks*).

En el contexto de cargas de trabajo multiprogramadas, estas limitaciones del parche `SCHED_HMP` dan lugar a una gran variabilidad en los tiempos de ejecución de los programas en distintas ejecuciones. Para ilustrar este problema, se procedió a realizar el siguiente experimento con una carga de trabajo formada por cuatro aplicaciones de SPEC CPU (`galgel`, `gamess`, `hmmer` y `povray`). Se realizaron 8 ejecuciones de la carga en un sistema asimétrico de ARM formado por dos cores Cortex A15 de alto rendimiento y dos cores Cortex A7 de bajo consumo. En cada ejecución se varió el orden de lanzamiento de las distintas aplicaciones y se mantuvo la carga de trabajo continua durante 20 minutos (lanzando varias instancias de cada benchmark si era necesario). La figura 1.1 muestra los tiempos de ejecución medios de cada benchmark en cada ejecución. Nótese que los

tiempos están normalizados con respecto a la ejecución más lenta de cada benchmark a lo largo de las ocho ejecuciones. Como se observa en la figura, una aplicación puede sufrir una gran variación en su tiempo de ejecución (hasta un 30 %) respecto a otras ejecuciones bajo SCHED_HMP.

Cabe destacar que esa variabilidad no está presente en el caso de algoritmos de planificación para sistemas asimétricos propuestos por la comunidad científica [3, 15, 34, 31]. En particular, la figura 1.2 muestra los resultados de la misma carga de trabajo considerada anteriormente pero usando un algoritmo que realiza un reparto justo del uso de los cores de alto rendimiento entre aplicaciones. Dicho algoritmo recibe el nombre de asymmetry-aware Round-Robin, pero por simplicidad se hará referencia a este algoritmo como RR. Los resultados de la figura 1.2 revelan que bajo RR los tiempos de ejecución de las aplicaciones son muy similares entre ejecuciones, que es lo que el usuario cabría esperar. Debido a que el planificador SCHED_HMP no garantiza tiempos de ejecución repetibles, los valores de las distintas métricas de rendimiento/justicia no son representativas y pueden dar lugar a conclusiones erróneas. Por este motivo, en el análisis experimental realizado en este Trabajo Fin de Grado, no consideramos los resultados del planificador SCHED_HMP.

1.3. Objetivos del proyecto

A pesar de que los principales fabricantes de procesadores están apostando por los AMPs, aún no existe un soporte robusto de planificación en Linux para este tipo de arquitecturas. El objetivo fundamental del proyecto es evaluar implementaciones en Linux de los algoritmos de planificación más relevantes para AMPs propuestos por la comunidad científica. Gran parte de estos algoritmos se encuentran ya implementados en el *framework de planificación* para Linux, que ha sido desarrollado por algunos miembros del grupo de Investigación ArTeCS de la UCM. Cabe destacar que el *framework de planificación* es bastante complejo (unas 25 mil líneas de código) y hasta el inicio de este TFG solo se había experimentado con él en arquitecturas x86, como el prototipo QuickIA de Intel [4]. En este proyecto ha sido necesario llevar a cabo una adaptación de ese *framework de planificación* a una versión del kernel soportada por la plataforma asimétrica elegida para nuestro estudio: la placa Odroid XU-4. Esta placa de desarrollo integra un procesador big.LITTLE de ARM. En el Capítulo 2 se pueden encontrar más detalles acerca de este sistema.

La eficiencia energética es uno de los aspectos clave de los distintos algoritmos que se pretende evaluar. Lamentablemente, la plataforma sobre la cual se llevará a cabo la evaluación no está provista de registros o sensores para que el software de sistema obtenga información en tiempo de ejecución sobre el consumo de potencia o de energía. Para superar esta limitación, se ha empleado un medidor de consumo externo que también actúa como fuente de alimentación de la placa Odroid XU-4. A pesar de la existencia de software para obtener medidas en tiempo real del medidor de consumo, el fabricante del dispositivo no proporciona un driver a nivel de kernel del SO para acceder a dicha información. Este soporte es necesario para obtener medidas de consumo a grano fino en nuestro estudio

experimental. Por lo tanto, durante el proyecto se ha procedido a desarrollar un driver a nivel de kernel en Linux. El driver desarrollado se encuentra actualmente integrado en la rama oficial de la herramienta de monitorización del rendimiento PMCTrack [26], usada ampliamente durante el desarrollo de este TFG.

1.4. Plan de trabajo

A principio del curso académico 2015/2016 realizamos una reunión preliminar en la que definimos los objetivos presentados en la sección anterior y establecimos una metodología de trabajo formada por las siguientes pautas:

- Establecer canales de comunicación (Google Drive, Git y servicios de correo electrónico), hábitos de reuniones y reparto de trabajo.
- Realizar una evolución incremental de la carga de trabajo y la frecuencia de las reuniones a lo largo del curso.
- Mantener vías de trabajo paralelas en la medida de lo posible, informando siempre de los resultados y técnicas empleadas.
- Muchas fases importantes se realizaron en equipo debido a la importancia de su conocimiento por ambos miembros.

La planificación que establecimos contempla las siguientes tareas:

1. Labor de investigación sobre las diferentes herramientas utilizadas durante el TFG (PMCTrack, PALLOC, framework de planificación, etc.) y sobre los conceptos necesarios para abordar los desarrollos y la evaluación experimental (*drivers* de dispositivos USB, compilación cruzada del *kernel* Linux, etc.)
2. Configuración de la plataforma de experimentación que utilizaremos (Odroid XU4).
3. Desarrollo del driver USB en Linux para obtener medidas de consumo de energía del medidor externo
4. Análisis offline para caracterizar las aplicaciones SPEC CPU que usaremos en los experimentos.
5. Análisis teórico para evaluar el potencial de distintos algoritmos a la hora de optimizar el rendimiento, la justicia o la eficiencia energética.
6. Adaptación del *framework de planificación* a la versión del kernel más adecuada con soporte para la placa Odroid XU-4
7. Evaluación de los distintos algoritmos de planificación usando cargas de trabajo multiprogramadas y discusión de resultados obtenidos
8. Diseño de modelos de estimación basados en contadores *hardware* para aproximar ratios de rendimiento y eficiencia energética relativa entre cores de la plataforma asimétrica

El orden de las tareas que figura en el anterior listado es meramente orientativo. Si bien fue posible realizar algunas de estas tareas en paralelo, como las tareas 3 y 4. Sin embargo, en ciertas etapas del TFG no se pudo explotar eficazmente el paralelismo a nivel de estudiante debido a las limitaciones del hardware, dependencias entre tareas y la obligatoriedad de ser conscientes de los conocimientos aplicados.

1.5. Estructura de la memoria

La memoria del proyecto se organiza en los siguientes capítulos:

- El **Capítulo 2** introduce el conjunto de dispositivos y herramientas software empleados para configurar el entorno experimental e implementar el código requerido en este proyecto.
- El **Capítulo 3** describe los principales algoritmos de planificación y métricas que han sido caso de estudio en nuestro proyecto.
- El **Capítulo 4** presenta los resultados obtenidos en base a la ejecución de diferentes *benchmarks* y cargas de trabajo usando los algoritmos descritos en el capítulo anterior.
- El **Capítulo 5** expone las conclusiones finales de este Trabajo de Fin de Grado y discute posible trabajo futuro. Se presenta un modelo para estimar diferentes métricas en tiempo de ejecución basado en los resultados obtenidos.
- Finalmente, se proporcionan varios apéndices. En ellos se incluye: (A) Introducción y (B) Conclusiones del Trabajo de Fin de Grado traducidas al inglés. Por último: (C) Contribuciones de cada participante al proyecto.

Capítulo 2

Entorno experimental

En este capítulo se presentan las características de los dispositivos hardware y herramientas software empleadas durante el TFG. Concretamente, en la Sección 2.1 se describen las características de la placa de desarrollo utilizada para llevar a cabo nuestra evaluación experimental. Más adelante, en la Sección 2.2 se presentan las principales herramientas utilizadas. Finalmente, en la Sección 2.3 se describe el entorno experimental de medida de consumo utilizado en este proyecto, que se basa en el dispositivo Odroid Smart Power.

2.1. Placa Odroid XU4

Para la realización del Trabajo de Fin de Grado se ha utilizado la placa de desarrollo Odroid XU4 [11], que se muestra en la figura 2.1. Esta placa está provista de un SoC (*System-On-Chip*) Exynos 5422 de Samsung fabricado en 28nm, que integra un procesador ARM big.LITTLE de ocho núcleos. Las características del procesador y de la jerarquía de memoria de esta plataforma se resumen en la tabla 2.1. De forma adicional, la figura 2.2 muestra el diagrama de bloques de la placa Odroid XU4.

Al comienzo del proyecto se instaló Ubuntu 15.04 en una tarjeta MicroSD que se conectaría a la placa. En primer lugar, partimos de una imagen ya creada del sistema operativo que proporciona el fabricante de la placa. Para interactuar con el sistema instalado, la placa ofrece dos vías fundamentales de comunicación, el puerto serie y la conexión Ethernet. Durante el transcurso del TFG, hemos utilizado la comunicación vía puerto serie desde una *workstation* empleando el programa `minicom`. A diferencia de la

Modelos de core	Número de cores / Pipeline	Último nivel de cache	Memoria principal
Cortex A15 @ 2.0GHz Cortex A7 @ 1.4GHz	4 / Ejec. fuera de orden 4 / Ejec. en orden	2MB (L2) 512KB (L2)	2GB DDR3 @ 750MHz

Tabla 2.1: Características del procesador y de la jerarquía de memoria de la placa Odroid XU4

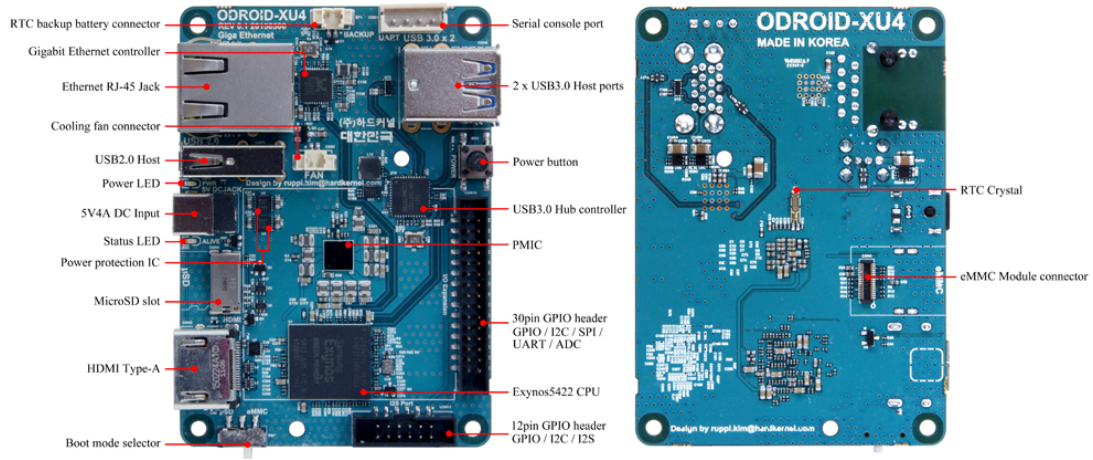

Figura 2.1: Placa ODROID-XU4

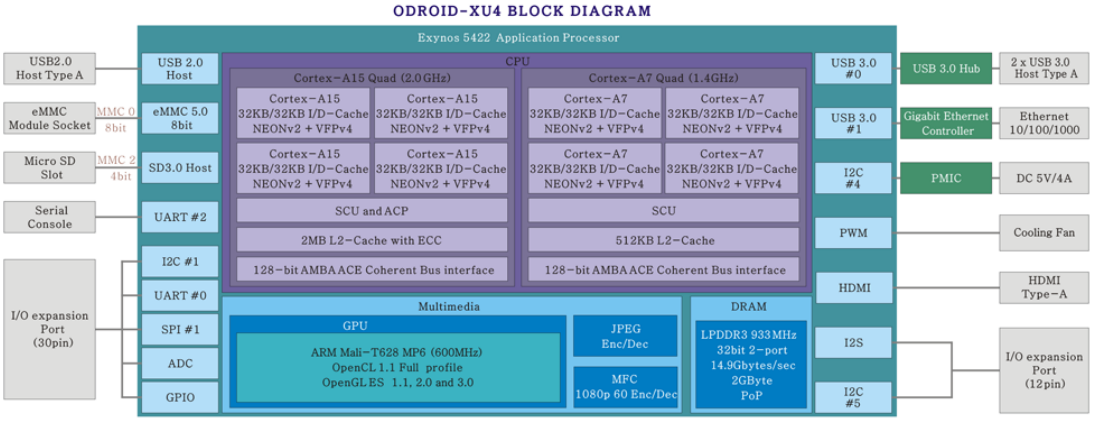

Figura 2.2: Diagrama de bloques de la placa Odroid XU4

conexión por red, la comunicación serie permite interactuar con el gestor de arranque (`uboot`), instalado junto al SO. Esto permite, entre otras cosas, seleccionar de forma remota la versión del kernel que se iniciará. En otras tareas, como el lanzamiento de experimentos o la edición de ficheros, se ha usado la conexión a través de la red (vía SSH).

2.1.1. Problemática versión del kernel

Como se mencionó en el capítulo de Introducción, una de las tareas más críticas del proyecto es la adaptación del Framework de Planificación para sistemas asimétricos a la placa Odroid XU4. Al inicio del TFG, sólo existían tres versiones de dicho framework: para las versiones 2.6.39, 3.2 y 3.17 del kernel Linux, respectivamente. Lamentablemente, el soporte para la placa Odroid XU4 sólo está disponible para *kernels* de versión 3.10 en adelante. Para simplificar el complejo proceso de adaptación del framework a la placa, se optó en primer lugar por instalar una variante del kernel 3.17 con soporte para Odroid XU3[1]. Durante el proceso de validación de ese kernel, detectamos que el driver CPU-freq proporcionado no funcionaba correctamente en la plataforma. En consecuencia, los cores de la placa no podían configurarse a la máxima frecuencia soportada lo que resultaba un gran impedimento para llevar a cabo el estudio experimental. A pesar de que se procedió a evaluar el soporte de otras versiones del kernel cercanas a la 3.17, no encontramos ninguna que proporcionase el soporte requerido.

Estas dificultades nos obligaron a utilizar una versión modificada del kernel 3.10.96 proporcionada por el fabricante, que sí dispone del soporte de todas las características requeridas por nuestro entorno experimental: driver CPU-freq, driver para el ventilador, parche `SCHED_HMP` para el planificador, etc. Usar este kernel garantiza el correcto funcionamiento del entorno experimental deseado, pero a costa de tener que portar el código del Framework de Planificación (unas 25000 líneas de código) a esta versión del núcleo. En el siguiente capítulo se puede encontrar más información acerca del Framework de Planificación.

Mientras se trabajaba en la migración del código del Framework de Planificación a la versión del kernel escogida, empleamos la placa para llevar a cabo experimentos de caracterización del rendimiento de distintos *benchmarks*. Para llevar a cabo estos experimentos, usamos la herramienta PMCTrack [26], que se presenta en la sección 2.2.1. La configuración de esta herramienta demandó intrucir otro parche en el kernel. En definitiva, durante este TFG se utilizaron dos versiones del kernel para la placa Odroid XU4:

- *3.10.96-pmctrack*: Esta versión incluye soporte para la herramienta de gestión de contadores hardware PMCTrack y para realizar medidas de consumo energético usando el software desarrollado en este TFG.
- *3.10.96-linux-amp*: Esta versión del kernel incluye el código del Framework de Planificación más el siguiente soporte:

[1]El código fuente de esta versión del kernel se encuentra en `https://github.com/abhijeet-dev/linux-samsung`

- Extensiones para la herramienta PMCTrack
- Soporte de particionado de cache vía software mediante PALLOC(ver sección 2.2.2)

2.2. Herramientas utilizadas

Para llevar a cabo nuestro proyecto se han empleado tres herramientas esenciales: PMCtrack, PALLOC y Het-Harness. A continuación se proporciona una descripción de cada una de ellas.

2.2.1. PMCTrack

PMCTrack es una herramienta de monitorización del rendimiento mediante contadores hardware para Linux[26]. Esta herramienta ha sido diseñada específicamente para ayudar a los desarrolladores del kernel en la implementación de algoritmos de planificación que utilizan los datos de los contadores de monitorización del rendimiento (*Performance Monitoring Counters* - PMCs) para realizar optimizaciones en tiempo de ejecución. A pesar de ser una herramienta orientada al sistema operativo, PMCTrack también está equipada con una serie de componentes en espacio de usuario (bibliotecas, entorno gráfico y de línea de comandos) que permiten obtener información de monitorización de aplicaciones en ejecución. En un trabajo previo [33] se discuten ampliamente las ventajas que PMCTrack ofrece frente a otras herramientas de monitorización del rendimiento.

La funcionalidad de PMCTrack puede ampliarse fácilmente mediante la creación de nuevos *módulos de monitorización*. Un módulo de monitorización es una extensión que se implementa mediante un módulo cargable del kernel y amplía la funcionalidad[2] de PMCTrack [33]. Los módulos de monitorización permiten exponer al usuario (y al planificador del SO) cualquier tipo de información de monitorización proporcionada por los procesadores modernos, pero que no esté modelada directamente a través de contadores hardware. Como el consumo de energía o el espacio que una aplicación utiliza en una cache compartida [26]. Esta información se expone a los distintos componentes de PMCTrack mediante una abstracción denominada "contadores virtuales".

El comando `pmctrack` es la forma más directa de acceder a la funcionalidad de PMCTrack desde modo usuario. Este comando soporta tres modos de uso:

1. ***Time-Based Sampling (TBS)***: Este modo permite obtener valores de los PMCs y contadores virtuales cada cierto tiempo (intervalo de muestreo) para una aplicación específica. Éste fue el modo de uso de `pmctrack` empleado durante nuestros experimentos de caracterización del rendimiento y consumo energético de distintas aplicaciones.

[2]Más información acerca de la creación de módulos de monitorización puede encontrarse en la página web oficial de PMCTrack [21]

2. ***Event-Based Sampling (EBS)***: En este modo la herramienta toma muestras del valor de los PMCs y los contadores virtuales para una aplicación específica cuando un determinado contador hardware alcanza un cierto umbral (indicado por el usuario).
3. ***Time-Based system-wide monitoring mode***: Este modo es una variante del modo TBS, donde la información de monitorización se proporciona por cada CPU (core) del sistema, en lugar de para una aplicación seleccionada. Este modo se activa mediante la opción `-S` de `pmctrack`.

Para ilustrar el funcionamiento de la herramienta `pmctrack`, consideremos el siguiente comando de ejemplo para el modo de TBS (por defecto):

```
$ pmctrack -c instr,cycles ./galgel00
[Event-to-counter mappings]
pmc1=instr
pmc2=cycles
[Event counts]
nsample    pid      event          pmc1          pmc2
      1  27204       tick    2247040326    1723742839
      2  27204       tick    2423705061    1957082929
      3  27204       tick    2466664612    1944385684
      4  27204       tick    2280669757    1964700454
      5  27204       tick    2595158266    1983839065
      6  27204       tick    2462262543    1980869030
      7  27204       tick    2474564037    1942807991
      8  27204       tick    2307308640    1978270869
      ...
```

Este comando proporciona al usuario el número de instrucciones retiradas y los ciclos de procesador por segundo. Para indicar los conjuntos de eventos hardware a monitorizar, es preciso utilizar la opción -c. Como se muestra en el ejemplo, la herramienta de línea de comandos permite especificar los eventos hardware a monitorizar usando mnemotécnicos, de la misma manera que otras herramientas orientadas a espacio de usuario [14, 5, 20]. Además, el programa soporta monitorización de aplicaciones tanto multihilo como monohilo y tiene capacidad de multiplexación de eventos [33]. El comienzo de la salida del comando muestra la asignación de cada evento hardware monitorizado a cada contador físico. En la salida, la sección *Event counts* muestra una tabla con el número de filas de los diversos eventos; cada muestra (una por segundo[3]) está representado por una fila diferente. Al final de la línea, se especifica el comando para ejecutar la aplicación que deseamos monitorizar (por ejemplo, ./galgel00).

2.2.2. PALLOC

En los sistemas *multicore*, el último nivel de cache y la DRAM son recursos compartidos críticos. En especial, en el contexto cargas de trabajo multiprogramadas que hacen un uso intensivo de la jerarquía de memoria. La contención en el último nivel de cache y la DRAM, puede llegar a convertirse en uno de los principales cuellos de botella del sistema [6, 41]. Este problema se amplifica por el hecho de que no todas las aplicaciones

[3]El período de muestreo de los contadores puede configurarse mediante la opción -T.

experimentan la misma degradación de rendimiento en situaciones de contención. Gran parte del problema reside en que las políticas de reemplazamiento de cache y planificación de peticiones a la DRAM no están diseñadas para mejorar la calidad de servicio, y en situaciones de contención pueden provocar una degradación significativa del rendimiento global [23, 6].

La herramienta PALLOC [40] permite mitigar los efectos de la contención de recursos compartidos. Esencialmente, PALLOC es un gestor de memoria para el kernel Linux, que permite realizar particionado de bancos de DRAM y de cache mediante técnicas de coloreado de páginas [37]. Esta herramienta se distribuye a través de un parche del kernel Linux que reemplaza al algoritmo básico de gestión de memoria (*buddy allocator*).

El procesador big.LITTLE integrado en la placa Odroid XU4 está formado por dos clusters de cores: un cluster de 4 cores *big* (Cortex A15), y otro cluster de 4 cores *little* (Cortex A7). Como se detalla en la sección 2.1, cada cluster de cores posee una cache de segundo nivel compartida. Estas caches compartidas de último nivel (2 Mb y 512 Kb, respectivamente para los cores *big* y *little*) son de un tamaño mucho más reducido que las que se pueden encontrar en los procesadores de alto rendimiento[4]. En experimentos preliminares, hemos observado que la contención derivada del tamaño reducido de la cache compartida entre clusters de cores provoca una degradación sustancial del rendimiento. En este contexto, PALLOC puede ser una herramienta de gran utilidad para aislar ciertas aplicaciones intensivas en memoria en un contexto de cargas de trabajo multiprogramadas. Hemos observado que particionar la cache con PALLOC resulta especialmente efectivo en los casos en los que, en la misma carga de trabajo, hay una aplicación intensiva en memoria y otra sensible a compartir la cache (se incrementan significativamente los fallos de cache y se degrada el rendimiento).

Para ilustrar el impacto de la contención cache realizamos una serie de experimentos con la ayuda de PMCTrack y PALLOC. El objetivo de nuestro análisis es caracterizar las aplicaciones según su grado de sensibilidad a compartir la cache con otras que se ejecutan en *cores* del mismo tipo. Para cuantificar el efecto de la contención procedimos a construir curvas MRC (*Miss-Rate Curves*). La MRC de una aplicación representa el número de fallos de cache (típicamente del último nivel) por cada mil instrucciones retiradas para distintos tamaños de cache. Algunos trabajos previos han demostrado que las curvas MRC resultan muy efectivas para estudiar el efecto de la contención [23, 37].

Mediante la construcción de MRCs para distintos *benchmarks* SPEC CPU pudimos observar comportamientos muy diferentes. Por ejemplo, la figura 2.3 muestra las MRCs obtenidas para los *benchmarks* `galgel` y `bzip` en el core *big* (Cortex A15). Como se puede observar, la tasa de fallos de cache para el benchmark `bzip` aumenta de forma en gran medida al reducir el tamaño de cache por debajo 1280 bytes. Por el contrario, para el benchmark `galgel`, es necesario reducir el tamaño de cache de forma significativa (por debajo de 512kB) para observar un aumento sustancial de la tasa de fallos. Estos

[4]Por ejemplo, el procesador Intel Xeon E3-1225C (quad-core), posee una cache compartida de último nivel de 8MB.

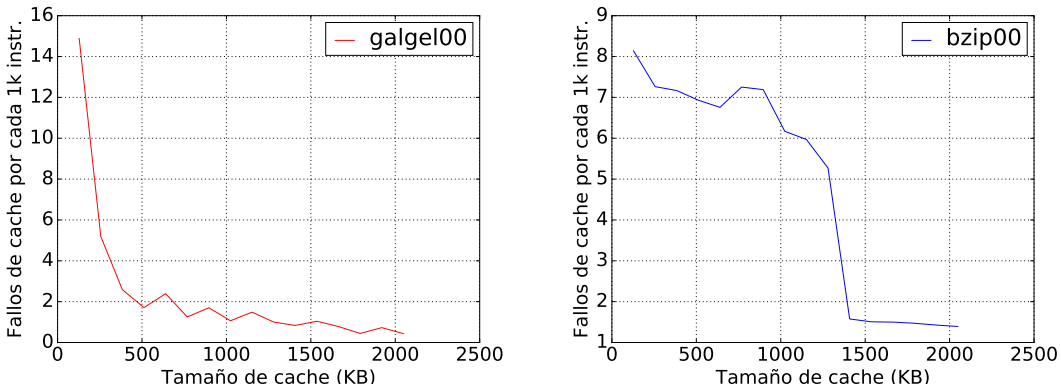

Figura 2.3: Relación entre tamaño de cache disponible y la tasa de fallos de LLC por cada mil instrucciones para los benchmarks galgel00 (izquierda) y bzip (derecha).

resultados revelan que `bzip` es más sensible a la contención cache que `galgel`: para observar degradación en el rendimiento de `bzip` es necesario una menor reducción en el tamaño de cache, que puede ocurrir como resultado de la compartición de la cache de último nivel con otras aplicaciones en una carga de trabajo multiprogramada.

En base a los resultados obtenidos en estos experimentos pudimos diferenciar dos tipos fundamentales de aplicación:

- Las aplicaciones **sensibles** a compartir cache. Cuando ven reducido mínimamente la cantidad de memoria disponible para su ejecución, se produce muy pronto un incremento de los fallos de cache de último nivel; degradando su rendimiento significativamente.

- Las aplicaciones **resistentes** a compartir cache. Sufren un incremento en la cantidad de fallos de cache solo cuando hay una disminución notable de la memoria disponible. Es decir, presentan un comportamiento menos sensible al uso compartido de la cache (en general se tratan de aplicaciones intensivas en CPU).

2.2.3. Het-Harness

La herramienta Het-Harness es un conjunto de utilidades (scripts BASH y Python, programas en C y ficheros de configuración) dirigidas a facilitar el lanzamiento de aplicaciones o cargas de trabajo multiprogramadas de manera sistemática. Actualmente, Het-Harness tiene soporte para GNU/Linux, Solaris y Mac OS X.

Het-Harness posee dos lanzadores de cargas de trabajo multiprogramadas: `run_benchsets_default` y `run_benchsets_amp`. El primer lanzador se emplea para lanzar cargas de trabajo con el planificador por defecto del SO (p.ej., CFS en Linux y TS en Solaris). El segundo lanzador permite lanzar distintas combinaciones de programas sobre la clase de planificación AMP, que implementa el *Framework de Planificación* usado

en este Trabajo de Fin de Grado. Ambos lanzadores tienen dos aspectos en común: (1) generan *logs* muy detallados con información de tiempos y consumo energético (en plataformas soportadas) de cada carga de trabajo y (2) reconocen ficheros de descripción de cargas de trabajo utilizando definiciones de alto nivel.

Los ficheros de descripción de cargas trabajo en Het-Harness son scripts BASH con la extensión ".spec". En un fichero de este tipo las distintas cargas y comandos de cada carga se definen empleando la notación de arrays de BASH. A continuación, se incluye un ejemplo de fichero de definición de cargas de trabajo:

```
#Command-line associated with each application
gobmk06=(GOBMK06)
h264ref06=(H264REF06)
gamess06=(GAMESS06)
bodytrack_p=(BODYTRACK_P 6)
semphy_mb=(SEMPHY_MB 6 active)
blast=(BLAST 12)
swim_m=(SWIM_M 7 spin)
wupwise_m=(WUPWISE_M 6 spin)
applu_m=(APPLU_M 12 spin)
#Workload composition
exp1=(wupwise_m bodytrack_p gobmk06)
exp2=(blast h264ref06)
exp3=(swim_m semphy_mb)
exp4=(applu_m gamess06)
#Test Vector
test_vector=(exp1 exp2 exp3 exp4)
```

Este fichero define cuatro cargas de trabajo formadas por distintas aplicaciones. La primera sección del fichero describe el comando que será utilizado para lanzar cada uno de los *benchmarks* de las distintas cargas. La segunda sección (a partir del comentario `#Workload composition`) define las cargas de trabajo (exp1, exp2, exp3, exp4). La última sección es el array de cargas de trabajo (`test_vector`).

Para poder usar un *benchmark* en un fichero de descripción de cargas, éste tiene que haber sido declarado previamente en el inventario de *benchmarks* de Het-harness. Este inventario, no es más que un fichero de texto donde para cada *benchmark* se definen dos aspectos:

1. **Identificador para el benchmark**: cadena de texto con letras mayúsculas, números y '_'
2. **Script**: ruta de un *script* de lanzamiento para el *benchmark*, que puede aceptar parámetros.

Los lanzadores de cargas de trabajo unidos a la representación de alto nivel de conjuntos de cargas en Het-Harness, permiten crear *scripts* intuitivamente para lanzar múltiples experimentos bajo diversas condiciones (algoritmos de planificación, parámetros de los algoritmos, configuraciones de frecuencias de cores, etc.). Usar este entorno ha facilitado enormemente nuestro análisis experimental.

Otras características destacables de Het-harness son las siguientes:

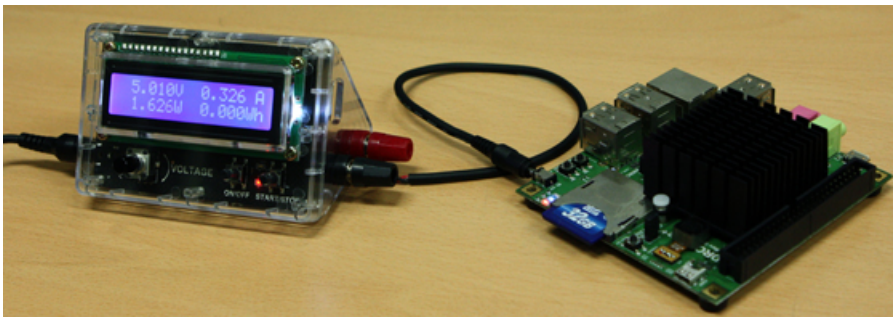

Figura 2.4: ODROID Smart Power

- Soporte para lanzamiento de *benchmarks* ampliamente usados como SPEC CPU2006, CPU2000, OMP2001 o PARSEC. La distribución básica de Het-harness ya proporciona scripts de lanzamiento para estos *benchmarks*.

- Scripts de procesamiento de los *logs* generados por los lanzadores de cargas de trabajo para obtener métricas de alto nivel (justicia, productividad, consumo energético, etc.)
- Integración con PMCTrack [26] y perf [39] para la monitorización de eventos hardware en la ejecución de distintos *benchmarks*.

2.3. Entorno de medida de consumo

Existen placas de desarrollo con procesadores ARM big.LITTLE que integran registros o sensores para obtener medidas de consumo de potencia instantánea o energía consumida a lo largo del tiempo para distintos componentes del SoC, como los *clusters* de cores, la memoria o la GPU integrada. Este es el caso de la placa de desarrollo ARM Juno o la Odroid-XU3+E. La placa ARM Juno tiene un coste mucho más elevado que la Odroid XU4: 5400 dólares vs. 80 dólares. El otro modelo de placa Odroid mencionado (XU3+E) ya no se comercializa, por lo que fue imposible adquirir esta plataforma para nuestro análisis experimental.

La principal limitación de la placa Odroid XU4 es que no está provista de estos sensores de medida de consumo, por lo ha sido preciso diseñar un entorno de medición específico sobre ella. El fabricante de la placa, también comercializa un dispositivo auxiliar llamado Odroid Smart Power, que se muestra en la figura 2.4. Este dispositivo es una fuente de alimentación para la placa que permite obtener medidas de potencia, corriente y voltaje instantáneos del sistema por USB. Asimismo, la placa está provista de una pantalla LCD donde se muestra en tiempo real el valor actual de las distintas medidas proporcionadas.

En este proyecto se desarrolló un *driver* de Odroid Smart Power para poder obtener medidas de consumo de potencia y energía desde la propia placa, conectada por USB al dispositivo. A pesar de que el fabricante del medidor de consumo ya proporciona software para obtener medidas de forma sencilla, este software está implementado como un programa de usuario (mediante *libusb*) y no como un *driver* en espacio de kernel. Acceder a las medidas de consumo en espacio de kernel es necesario en el contexto de nuestro proyecto para poder integrar el driver en la herramienta PMCTrack. Esto permite obtener para una aplicación medidas de consumo de potencia y energía a lo largo del tiempo obtenidas en sincronía con la información de rendimiento proporcionada por los contadores hardware del procesador.

En el resto de esta sección se describe el proceso de desarrollo y la funcionalidad de las distintas variantes del driver construido para el dispositivo Odroid Smart Power. Más adelante, se presenta el mecanismo usado para medir consumo en distintos escenarios requeridos por el análisis experimental realizado en este TFG.

2.3.1. Driver Odroid Smart Power

El proceso de desarrollo del *driver* para el dispositivo Odroid Smart Power se dividió en dos etapas. En la primera etapa, se construyó un driver *standalone* para el dispositivo (desacoplado de la herramienta PMCTrack). En la segunda etapa, se procedió a crear una versión ampliada del driver que se integra en PMCTrack mediante un módulo de monitorización para esta herramienta. El segundo driver, empleado en nuestros experimentos, exporta *contadores virtuales* de PMCTrack que facilitan la obtención de medidas combinadas de consumo y rendimiento mediante el comando `pmctrack` de modo usuario.

2.3.1.1. Driver *standalone*

Odroid Smart Power es un dispositivo USB que posee dos *endpoints* USB: uno de control, presente en todos los dispositivos USB; y un *endpoint* adicional de tipo *interrupt*, que controla las transferencias de datos desde/hacia el dispositivo. El comando `usb-devices` permite obtener la información específica del dispositivo, como el *Vendor ID*, *Product ID*, el número de *endpoints*, etc:

```
$ usb-devices
.....
T:  Bus=02 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#=  2 Spd=12  MxCh= 0
D:  Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=04d8 ProdID=003f Rev=00.02
S:  Manufacturer=Microchip Technology Inc.
S:  Product=Simple HID Device Demo
C:  #Ifs= 1 Cfg#= 1 Atr=c0 MxPwr=100mA
I:  If#= 0 Alt= 0 #EPs= 2 Cls=03(HID  ) Sub=00 Prot=00 Driver=spower
```

Para crear el driver *standalone* fue necesario emplear la *API* asíncrona que ofrece el USB core de Linux. Esta API ofrece control absoluto sobre los mensajes USB (conocidos como URB o *Usb Request Blocks*), que se envían al dispositivo USB y se reciben a través del driver. Cabe destacar que la controladora de *host* USB es la encargada en última instancia de realizar el envío y recepción de dichos URBs. No obstante, el API del USB core de Linux envuelve la interacción de bajo nivel con la controladora del *host* USB subyacente, así como el manejo de las interrupciones asociadas.

El driver *standalone* fue desarrollado en una estación de trabajo conectada por USB al dispositivo Odroid Smart Power, que a su vez alimentaba la placa Odroid XU4. El Odroid Smart Power se expone al SO como un dispositivo HID (*Human Interface Device*). Para gestionar este tipo de dispositivos, el kernel Linux destina un *driver* genérico llamado `usbhid`. Esto supone un problema, ya que por defecto no es posible asignar un driver específico al dispositivo Odroid Smart Power, al existir otro *driver* que ya lo gestiona. Para solucionarlo, fue necesario introducir la tupla (`vendor_id`,`product_id`) que identifica al Odroid Smart Power en la lista negra asociada al driver *usbhid*. Todo dispositivo incluido en esa lista negra es ignorado automáticamente por dicho driver. Para incorporar una nueva entrada a la lista negra del driver *usbhid*, dicho driver exporta un parámetro configurable en tiempo de carga llamado `quirks`. Este parámetro acepta valores con la siguiente sintaxis: `<vendor_id>:<product_id>:<command_id>`. Donde el comando con ID=4 permite incorporar el dispositivo USB en la lista negra. En definitiva, para lograr que `usbhid` ignore el dispositivo Odroid Smart Power es preciso usar el parámetro `quirks` con el valor "0x04d8:0x003f:0x0004". Si el driver se ya encuentra cargado en en el sistema, será preciso descargarlo y cargarlo de nuevo como sigue:

```
$ sudo rmmod usbhid
$ sudo modprobe usbhid quirks=0x04d8:0x003f:0x0004
```

En caso de que el driver `usbhid` no haya sido compilado como módulo cargable sino incluído en el binario monolítico del kernel, será necesario incluir el siguiente *flag* en el comando de arranque del kernel Linux:

```
usbhid.quirks=0x04d8:0x003f:0x0004
```

La funcionalidad del driver *standalone* desarrollado en primera instancia es muy reducida. Dicho driver expone el fichero de dispositivo `/dev/usb/spower0`, que se crea automáticamente al conectar el Odroid Smart Power al sistema cuando el driver está cargado. Ese fichero de dispositivo permite al usuario obtener las medidas de voltaje, corriente, y potencia instantánea proporcionadas por Odroid Smart Power realizando una lectura desde el terminal con el comando `cat`. A continuación se muestra un ejemplo:

```
$ cat /dev/usb/spower0
4750mV, 843mA, 3992mW
4750mV, 843mA, 3992mW
4750mV, 997mA, 4713mW
4750mV, 998mA, 4719mW
4750mV, 999mA, 4725mW
4750mV, 1001mA, 4732mW
4750mV, 1001mA, 4736mW
4750mV, 1002mA, 4740mW
4750mV, 1002mA, 4742mW
```

Cada línea de la salida se corresponde con una lectura, tomada cada 200ms. Para modificar el periodo de muestreo se puede escribir el nuevo valor (especificado en milisegundos) en el fichero de dispositivo. Por ejemplo, para emplear 100ms como nuevo intervalo de muestreo se debería ejecutar el siguiente comando: `echo 100 > /dev/usb/spower0`.

2.3.1.2. Driver integrado en PMCTrack

Como se mencionó en la sección 2.2.1, la herramienta PMCTrack permite implementar una serie de mecanismos para extender su funcionalidad, llamados *módulos de monitorización*. Estos módulos permiten incorporar soporte para medir diferente tipo información (p.ej., consumo energético) que no se expone al SO mediante la interfaz de contadores hardware. Los módulos exponen la nueva información de monitorización al usuario mediante una abstracción que introduce PMCTrack denominada *contadores virtuales*. Una discusión completa sobre el potencial de los módulos de monitorización y los contadores virtuales de PMCTrack puede encontrarse en [33, 21].

Nuestro objetivo en esta segunda fase del desarrollo fue integrar el driver *standalone* en PMCTrack y desarrollar un módulo de monitorización que permitiera exportar las medidas de consumo proporcionadas por Odroid Smart Power mediante contadores virtuales. Para ofrecer este soporte fue preciso realizar modificaciones sustanciales en el driver original. Uno de los principales problemas, era garantizar que la monitorización simultánea de múltiples aplicaciones siendo posible, a pesar de disponer de una única fuente de medida de consumo energético. Para ello, se extendió el driver con un API sencilla y consciente de la concurrencia (*SMP-safe*) que permite obtener desde varios puntos del código y de forma asíncrona las medidas más recientes proporcionadas por el dispositivo Odroid Smart Power.

La implementación de esta API, hace uso de un buffer circular acotado. Cada vez que el driver USB obtiene una nueva muestra del dispositivo, la inserta en el buffer circular y asocia a dicha muestra un *timestamp*. El buffer almacena únicamente las k últimas muestras obtenidas. Cada vez que la herramienta PMCTrack solicita una muestra, se invoca una función del driver a la que se pasa como parámetro un *timestamp* (T), que indica cuándo se solicitó la muestra anterior en esta sesión de monitorización. En respuesta a esta solicitud, el driver retorna una muestra agregada, obtenida mediante el promedio de las muestras almacenadas en el buffer que se recabaron a partir de T.

En resumen, el nuevo driver USB junto con el módulo de monitorización desarrollado proporcionan la siguiente funcionalidad:

- Capacidad de configurar la frecuencia de obtención de muestras del dispositivo Odroid Smart Power mediante el sistema de ficheros /proc. Nótese que durante una sesión de monitorización con PMCTrack no es necesario utilizar el mismo periodo de muestreo, gracias al API asíncrona proporcionada por el nuevo driver.

- La información proporcionada por el Odroid Smart Power se exporta mediante 3 contadores virtuales: `power_mw`, `current_ma` y `energy_uj`. El último contador virtual indica la energía consumida durante el periodo de muestreo usado en la monitorización con PMCTrack, este valor que se calcula a partir de las múltiples muestras de consumo de potencia obtenidas en el intervalo de muestreo.

El soporte de monitorización de consumo energético para el dispositivo Odroid Smart Power desarrollado durante este TFG se encuentra integrado en la rama oficial de PMCTrack [22]. Para ilustrar el uso del software desarrollado mostramos a continuación los pasos a seguir para monitorizar el rendimiento y el consumo energético de una aplicación en la plataforma experimental. El único requisito para llevar a cabo estos pasos es tener instalada la última versión de PMCTrack en el sistema. Para cargar el nuevo módulo de monitorización que obtiene medidas de consumo, debemos cargar primero el módulo del kernel de PMCTrack para la placa Odroid XU4 (`mchw_odroid_xu.ko`). Tras conectar el dispositivo Odroid Smart Power por USB a la placa, debemos asegurarnos de que el driver integrado en PMCTrack ha detectado correctamente el dispositivo. Para ello ejecutaremos el siguiente comando:

```
$ cat /proc/pmc/mm_manager
    [*] 0 - This is just a proof of concept
    [ ] 1 - IPC sampling SF estimation module
    [ ] 2 - Odroid Smart Power
    [ ] 3 - Oracle SF estimation model
    [ ] 4 - EDP Monitoring module
    [ ] 5 - Sched prototype monitoring module
```

Para activar el módulo de monitorización basta con teclear el siguiente comando:`$ echo activate 2 > /proc/pmc/mm_manager`. Una vez hecho esto podríamos comprobar que el módulo de monitorización se ha activado correctamente y además exporta los contadores virtuales citados anteriormente como sigue:

```
$ cat /proc/pmc/mm_manager
    [ ] 0 - This is just a proof of concept
    [ ] 1 - IPC sampling SF estimation module
    [*] 2 - Odroid Smart Power
    [ ] 3 - Oracle SF estimation model
    [ ] 4 - EDP Monitoring module
    [ ] 5 - Sched prototype monitoring module
$ pmc-events -V
[Virtual counters]
power_mw
current_ma
energy_uj
```

En este punto ya es posible iniciar una sesión de monitorización con PMCTrack para cualquier aplicación. El siguiente comando muestra el número de fallos de cache de último nivel, las instrucciones retiradas, el consumo de potencia medio (en mW) y consumo energético (en micro Julios) para el benchmark `hmmer` de SPEC CPU 2006 ejecutándose en la CPU #4 (core Cortex A15) de la placa Odroid XU4:

```
$ pmctrack -b 4 -c llc_misses,instr -V power_mw,energy_uj -T 0.5  ./hmmer06
[Event-to-counter mappings]
pmc1=llc_misses
```

```
pmc2=instr
virt0=power_mw
virt2=energy_uj
[Event counts]
nsample     pid       event          pmc1          pmc2     virt0         virt2
      1   13594        tick        537850    1747080239      4431       2215900
      2   13594        tick        667758    2050841122      6429       3214900
      3   13594        tick       1218564    2062162739      6764       3382200
      4   13594        tick       2164923    2038336610      6854       3427400
      5   13594        tick       1674625    2065411454      6936       3468400
      6   13594        tick       1862785    2059762222      6944       3472300
      7   13594        tick       1618331    2065219211      6942       3471300
      8   13594        tick       2384766    2032207379      7004       3502400
.....
```

2.3.2. Consumo de potencia neto

El driver del dispositivo Odroid Smart Power proporciona medidas agregadas de consumo de potencia y energía de todo el sistema. Nuestro objetivo es aproximar con la mayor precisión posible el consumo neto de los clusters de cores. Aislar los consumos específicos es un requisito para evaluar el consumo de potencia aplicaciones específicas y analizar los beneficios potenciales de distintos algoritmos de planificación en el contexto de cargas de trabajo multiprogramadas. En este escenario, las decisiones de planificación sólo afectan al consumo energético de los cores y de la memoria[5], y no al consumo de otros componentes del sistema como la GPU o la interfaz de red.

Asimismo, para caracterizar la eficiencia energética de una aplicación al ejecutarse en distintos tipos de cores (necesario para nuestro análisis experimental), es necesario poder aproximar el consumo energético neto resultante de ejecutar únicamente una aplicación en un core específico del sistema AMP. En este caso, el consumo neto debe incluir el consumo de la memoria y del cluster de cores (trabajando a máxima frecuencia) en el que la aplicación está ejecutándose.

Antes de presentar el modo de aproximar el consumo neto de potencia/energía, introducimos las siguientes definiciones:

- C_{big}: consumo de potencia del sistema en *idle* cuando los cores *big* operan a la máxima frecuencia (2GHz) y los cores *little* trabajan a la mínima frecuencia configurable (200Mhz).
- C_{small}: consumo de potencia del sistema en *idle* cuando los cores *small* operan a la máxima frecuencia (1.4Ghz) y los cores *big* trabajan a la mínima frecuencia configurable (200Mhz)
- C_{min}: consumo de potencia del sistema en *idle* cuando todos los cores del sistema (*big* y *small*) trabajan a la mínima frecuencia configurable (200Mhz)

[5]El consumo energético de la memoria que se observa cuando una aplicación secuencial se asigna a un core con ejecución fuera de orden (p.ej., Cortex A15) no es el mismo que cuando se ejecuta en un core con ejecución en orden (p.ej., Cortex A7). Esto se debe al distinto impacto de los fallos de cache durante la ejecución en ambos tipos de core.

Tabla 2.2: Fórmulas para obtener consumos netos de energía de una aplicación o carga de trabajo.

Ecuaciones
$C_{cluster(big)} = C_{big} - C_{min}$
$C_{cluster(small)} = C_{small} - C_{min}$
$C_{app(big)} = C_{system} - C_{cluster(big)}$
$C_{app(small)} = C_{system} - C_{cluster(small)}$
$C_{workload} = C_{system} - C_{base(min)}$

Para obtener estas medidas de consumo en *idle* para las distintas configuraciones de frecuencia, empleamos el módulo de monitorización descrito en la sección anterior usando la herramienta `pmctrack` del siguiente modo:

```
$ pmctrack -V power_mw -T 1 sleep 120
[Event-to-counter mappings]
pmc1=instr
virt0=power_mw
[Event counts]
nsample    pid      event       virt0
      1  14190       tick        3738
      2  14190       tick        3725
      3  14190       tick        3725
      4  14190       tick        3725
      5  14190       tick        3724
      6  14190       tick        3724
      7  14190       tick        3724
      8  14190       tick        3726
      9  14190       tick        3725
     10  14190       tick        3723
     11  14190       tick        3723
...
```

Esencialmente, el comando anterior proporciona el consumo de potencia medio (en mW) cada segundo mientras se "ejecuta" la aplicación `sleep`, que en realidad se bloquea durante el número de segundos especificados como parámetro (120). Por tanto, esta aplicación no realiza ningún tipo de procesamiento. Como se puede apreciar, el consumo de potencia en *idle* permanece relativamente estable durante la monitorización. No obstante, para obtener un valor de referencia para el consumo en idle nos quedamos con el mínimo valor observado durante un intervalo de dos minutos. Este experimento se realizó para cada configuración de frecuencias anteriormente citada.

Cabe destacar que durante experimentos preliminares (caracterización de *benchmarks* individuales) observamos que el ventilador de la placa se activaba en intervalos de tiempo irregulares. La activación puntual del ventilador daba lugar a picos de consumo de potencia de +0.5W aproximadamente. Estas activaciones tienen lugar cuando la temperatura

del SoC alcanza cierto umbral, lo cual ocurría con frecuencia al atravesar una fase de ejecución muy intensiva en CPU. También observamos que el ventilador se desactivaba en otros momentos en los que la temperatura se situaba de nuevo por debajo del umbral (p.ej., en fases de ejecución intensivas en memoria). Para eliminar los picos de consumo de potencia, que ocurrían con distinta frecuencia dependiendo de la aplicación, optamos por cambiar el modo del ventilador de *automatic* a *ON*. De este modo, era posible eliminar la componente de consumo del ventilador y aislar el consumo de una aplicación específica con mayor precisión. Para ello, el consumo del ventilador se agregó a los consumos de los cores en *idle* definidos anteriormente.

La Tabla 2.2 presenta una serie de fórmulas que empleamos para calcular el consumo de potencia neto cuando una aplicación se ejecuta sola en el sistema empleando un cluster de cores específico (denotado como $C_{app(big)}$ y $C_{app(small)}$ respectivamente), y para calcular el consumo de potencia neto de una carga de trabajo multiprogramada ($C_{workload}$). En las fórmulas, C_{system} denota el consumo de potencia del sistema completo, proporcionado por el dispositivo Odroid Smart Power. Nótese que estas fórmulas nos permiten aproximar el consumo neto cuando una aplicación o carga de trabajo se ejecuta con todos los cores del sistema AMP configurados a la máxima frecuencia. Esta es la configuración utilizada durante los experimentos.

2.3.3. Medición de EDP

Una vez planteado el conjunto de fórmulas necesarias para aislar el consumo energético de las cargas de trabajo, debíamos hallar un método para calcular el producto energía retardo (EDP) resultante de la ejecución de una carga de trabajo. Como se discute en el siguiente capítulo, esto conlleva medir la energía neta consumida y el número de instrucciones por segundo de la carga. Para ello, se hizo uso de un módulo de monitorización auxiliar de PMCTrack. Este módulo mantiene tres contadores globales para cuantificar la información agregada del conjunto de *benchmarks* de la carga de trabajo:

- Instrucciones retiradas en cores big.
- Instrucciones retiradas en cores small.
- Energía consumida.

Cada vez que se lanza una nueva aplicación en el sistema con *Het-Harness*, el nuevo módulo de monitorización configura los contadores hardware del procesador para medir instrucciones retiradas en ambos tipos de core. Cuando la aplicación finaliza, se incrementan los contadores globales de instrucciones anteriormente citados.

El contador de energía consumida se actualiza cada vez que el driver de Odroid Smart Power obtiene una nueva muestra del dispositivo. El módulo de monitorización permite que el lanzador de cargas de trabajo en Het-harness, reinicie el valor del contador de energía (y los de instrucciones) cuando desee escribiendo la cadena de caracteres "restart_edp" en el fichero `/proc/pmc/config`. Típicamente el lanzador de cargas de trabajo reinicia los contadores globales justo antes de lanzar la carga, y consulta su

valor al finalizarla. Para obtener el valor de los contadores globales basta con realizar una lectura de la entrada `/proc/pmc/config`. Una vez leídos los contadores de energía e instrucciones retiradas, el lanzador calcula el EDP considerando también el tiempo de ejecución de la carga de trabajo.

Capítulo 3

Métricas y Algoritmos de planificación

En este capítulo se presentan las métricas y algoritmos de planificación empleados en nuestro análisis experimental. En particular, la Sección 3.1 expone las métricas que caracterizan el comportamiento de hilos en base a diferentes propiedades. En la Sección 3.2 se definen las métricas utilizadas para estudiar la efectividad de diferentes algoritmos de planificación al ejecutar cargas de trabajo multiprogramadas. La Sección 3.3 presenta el funcionamiento del *framework de planificación*, utilizado para evaluar diferentes algoritmos en el kernel Linux. Finalmente, la Sección 3.4 proporciona una breve descripción de los algoritmos de planificación analizados.

3.1. Métricas de hilos

Los algoritmos de planificación considerados emplean dos métricas clave (el SF y el EEF) para realizar asignaciones de hilos a cores de forma efectiva en sistemas multicore asimétricos.

El factor de ganancia o SF (*Speedup Factor*) denota la mejora de rendimiento que un hilo (en una aplicación secuencial) experimenta al ejecutarse en un core *big* frente a uno *small* en un sistema multicore asimétrico. El SF se define formalmente como el ratio de las instruciones que el hilo ejecuta por segundo (IPS) en el core *big* frente al core *small*:

$$SF = \frac{IPS_{Big}}{IPS_{Small}} \tag{3.1}$$

Para cuantificar la eficiencia energética que un hilo deriva de un core *big* frente a uno *small* en un sistema asimétrico, se ha propuesto en trabajo previo [32] una métrica denominada *Energy-Eficiency factor* (EEF). El EEF se define como sigue:

$$EEF = \frac{SF}{NET\_EPI_{big}} \tag{3.2}$$

donde NET_EPI_{big} denota el ratio de energía neta por instrucción observado en un core big (core que típicamente exhibe un consumo de potencia muy superior al del core *small*). Los algoritmos EEF-Driven y ACFS, que se presentan en la sección 3.4, hacen uso de este factor a la hora de realizar un reparto de los ciclos de core *big* asignados a las distintas aplicaciones.

3.2. Métricas de cargas de trabajo

Para medir el rendimiento global que una carga de trabajo experimenta en un sistema AMP y empleando un algoritmo de planificación dado, estudios previos[30, 31] han empleado la métrica **Aggregate SPeedup** (ASP), que se define como sigue:

$$ASP = \sum_{i=1}^{n} \left(\frac{CT_{slow,i}}{CT_{sched,i}} - 1 \right) \tag{3.3}$$

donde **n** es el número de aplicaciones en la carga de trabajo, $CT_{sched,i}$ define el tiempo finalización de la aplicación **i** ejecutada con un planificador determinado, y $CT_{slow,i}$ es el tiempo de finalización de la aplicación **i** cuando se ejecuta en el AMP usando los cores *small*. La métrica ASP (cuanto mayor, mejor) refleja la eficiencia total que una carga deriva de los varios tipos de core cuando se ejecuta bajo un planificador determinado.

Investigación previa sobre la justicia en sistemas multicore simétricos [7, 19, 6] y asimétricos [27, 38] define un algoritmo de planificación como justo si las aplicaciones de igual prioridad en una carga de trabajo multiprogramada sufren la misma degradación en rendimiento por compartir el sistema respecto a cuando se ejecutan solas. Para representar esta noción de justicia, hemos empleado la métrica **unfairness** (cuanto menor, mejor), que se define del siguiente modo:

$$Unfairness = \frac{MAX(Slowdown_1, ..., Slowdown_n)}{MIN(Slowdown_1, ..., Slowdown_n)} \tag{3.4}$$

donde $Slowdown_i = \frac{CT_{sched,i}}{CT_{fast,i}}$, y $CT_{fast,i}$ denota el tiempo de ejecución de la aplicación i cuando se ejecuta sola en el AMP (con todos los cores *big* disponibles).

Para cuantificar la eficiencia energética de una carga de trabajo multiprogramada empleamos el **producto energía-retardo** o **EDP** (*Energy-Delay Product*) [13, 8], que se define del siguiente modo:

$$EDP = \frac{Total\_Energy\_Consumed}{Intructions\_Per\_Second} \tag{3.5}$$

3.3. Framework de planificación

El *framework de planificación* implementa diversos planificadores y los agrupa en una clase de planificación denominada AMP que puede incorporarse al kernel sin modificar el comportamiento del planificador por defecto. El algoritmo "activo" se establece mediante un parámetro configurable en espacio de usuario. Asimismo, muchos aspectos comunes de los algoritmos de planificación, como la frecuencia de activación del proceso de intercambio de hilos entre cores, se controlan mediante distintos parámetros. Para mostrar todos los parámetros de configuración disponibles en el *framework*, se puede utilizar el comando `amp_readpar`, proporcionado por la herramienta Het-Harness:

```
odroid@torvalds:~$ amp_readpar
*** READING PARAMETERS **
--AMP_SCHED_CONFIG--
        selected_credit_algorithm 0
        het_min_credit_assigment_period 40
        het_max_credit_assigment_period 300
        het_prop_rr_utility_threshold 0
        het_prop_dyn_utility_threshold 0
        het_credit_runaway_threshold 15
        het_timeslices_per_swap 11
        het_notify_fast_core_fractions 0
        num_par_serial_trans_allowed 4
        het_utility_mode 1
        het_low_utility_threshold -1
        load_balance 1
        het_big_small_perf_ratio 200
        deferred_balance_on_inactive 1
        tslice_idle 3
        balance_interval 125
        het_vruntime_threshold 6700
        het_jiffies_reset_het_vruntime 5
        het_selected_hcfs_variant 0 (FAIRNESS)
        het_unfairness_factor 100
        het_update_avg_freq 20
        het_use_average_speedup 0
        het_swap_random_period 20
--SCHED_GBL_CONFIG
        amp_ticks_per_tslice (AMP_TICKS_PER_TSLICE ) 25
        amp_het_credits_per_tick (AMP_HET_CREDITS_PER_TICK) 140
        ampa_het_proportional_sched_period (AMP_HET_TIMESLICES_PER_ACCT) 40
        **Parametro no configurable** AMP_HET_CREDITS_PER_TSLICE 3500
        **Parametro no configurable** AMP_HET_CREDITS_PER_ACCT 140000
        amp_het_credits_per_acct_ref (AMP_HET_CREDITS_PER_ACCT_REF) 420
*****
```

Para modificar el valor de un parámetro, se puede usar el comando `amp_writepar` de la siguiente forma: `$ amp_writepar "<nombre_parámetro> <valor>"`. El parámetro `het_selected_hcfs_variant` permite seleccionar el algoritmo de planificación *asymmetry aware* activo. Por defecto, su valor es 0, que se corresponde con el algoritmo ACFS. Los valores soportados para ese parámetro son 0-9 (hay 10 planificadores implementados). Las distinas opciones disponibles se encuentran en en el array `hcfs_variants_str`, definido en el código fuente del *framework* de planificación:

```
static const char* hcfs_variants_str[HCFS_NR_VARIANTS]=
    {"FAIRNESS",
```

```
    "PROPSP",
    "RR_APP",
    "RR_THREAD",
    "LI",
    "HSP",
    "EQP",
    "HSP_MT",
    "EF_DRIVEN",
    "PRIM"};
```

Por ejemplo, para activar el planificador HSP, que es el que maximiza el rendimiento global, se ha de ejecutar el siguiente comando:

```
$ amp_writepar "het_selected_hcfs_variant␣5"
```

3.3.1. Migración de clase de planificación AMP

Como se explicó en la Sección 2.1.1, el *framework de planificación* estaba implementado sobre versiones del kernel que carecen de soporte para la placa Odroid XU4. Este hecho obligó a migrar el código del *framework* de planificación a la versión del kernel proporcionada por el fabricante de la placa (variante de Linux 3.10.96). La migración del código del *framework*, que se compone de alrededor de 25 mil líneas, supuso una tarea compleja. Además, fue necesario aplicar parches adicionales sobre el kernel para incluir el soporte necesario para PALLOC [40] y para PMCTrack [26, 33].

Para incorporar múltiples algoritmos de planificación conscientes de la asimetría el *framework de planificación* requiere la creación de una nueva clase de planificación en el kernel, denominada AMP. Por lo tanto, el primer paso en el proceso de migración del código del *framework* a la versión del kernel soportada por la placa, fue crear una nueva clase de planificación en el kernel Linux, que como base tenga el mismo comportamiento que la clase de planificación CFS. Para ello se realizó un clon de la misma haciendo las modificaciones necesarias.

Para comprender el proceso de creación del clon de CFS (base para la clase AMP), es preciso conocer algunos aspectos generales del diseño del planificador de Linux. Cada proceso tiene asociada una política de planificación. Los procesos que pertenecen a una determinada política son planificados por una clase de planificación. Por ejemplo, la clase de planificación RT, dedicada a procesos de tiempo real, implementa las políticas `SCHED_FIFO` y `SCHED_RR`. Por el contrario, la clase de planificación *fair* (algoritmo CFS) planifica los procesos con política `SCHED_NORMAL`. Se modificó el fichero `sched.h` que define estas políticas de planificación para añadir la nueva política implementada por la clase AMP (`SCHED_AMP`):

```
#define SCHED_NORMAL    0
#define SCHED_FIFO      1
#define SCHED_RR        2
#define SCHED_BATCH     3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE      5
#define SCHED_AMP       7
```

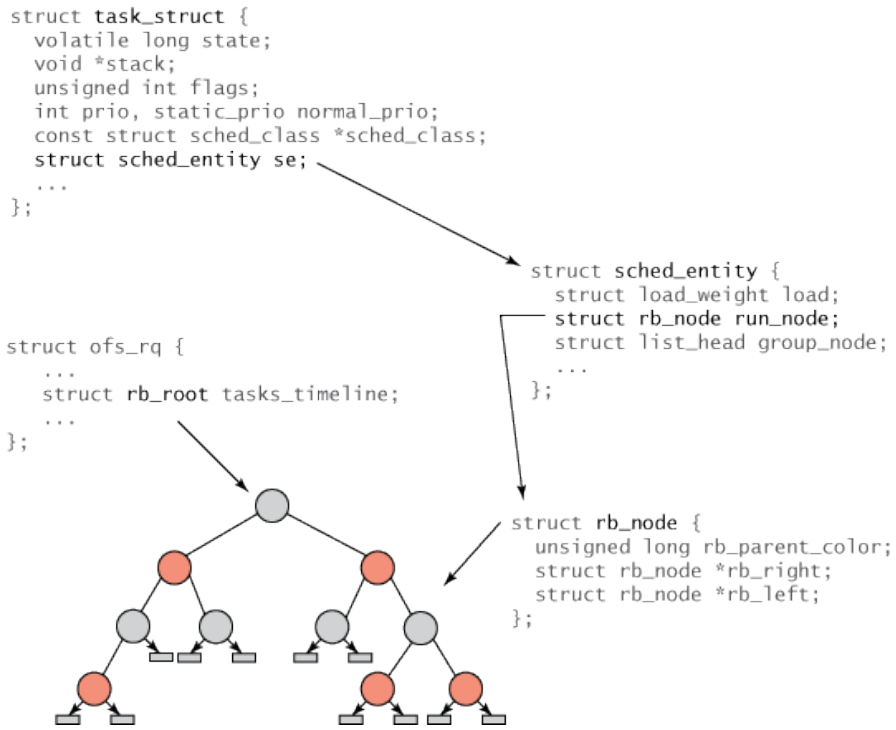

Figura 3.1: Jerarquía de estructuras *task_ struct* y *sched_ entity*

Las clases de planificación en el kernel se describen mediante una estructura (`struct sched_class`), que incluye un conjunto de punteros a funciones (operaciones básicas del planificador). Además, la estructuras que representan a las distintas clases están unidas formando una lista enlazada de clases de planificación que se puede navegar por medio del puntero `next` de la estructura. Para incluir la clase AMP, fue necesario registrar la estructura de la nueva clase de planificación AMP en esta lista enlazada. Para ello, se actualizaron las referencias de *Real Time* y la propia *AMP*. La lista quedó como sigue: Clase RT -> Clase Fair (CFS) -> Clase AMP -> Clase Idle -> NULL.

Los procesos dentro del kernel están representados mediante una estructura llamada `task_struct`. En su interior, cada proceso contiene una entidad (`sched_entity`) por cada clase de planaificación. La entidad de cada clase almacena campos utilizados únicamente por la clase de planificación en cuestión. Cabe destacar que la entidad es lo que se inserta realmente en las *run queues* del planificador y no el `task_struct`. Para poder usar una nueva clase de planificación, se definió una entidad `sched_amp_entity`. Posteriormente se añadió un campo de este tipo de datos en la estructura `task_struct` para que el planificador pueda almacenar la información necesaria de cada proceso. En la Figura 3.1 se muestra un esquema, que ilustra la interrelación entre las distintas estructuras anteriormente para el planificador CFS.

El planificador de Linux posee una *run queue* para cada CPU, donde se encolan los procesos que están listos para ejecutarse en el sistema. Notesé que cada *run queue*, se compone a su vez de otras *sub run queues*, para cada clase de planificación. Esto obligó a crear un tipo de datos `struct amp_rq` y a añadir un campo de ese tipo de datos a la estructura `struct rq`, que representa una *run queue*. El resto de los cambios realizados consistieron en actualizar nombres de funciones y variables, y actualizar ciertas condiciones para hacer al planificador de Linux consciente de la existencia de la nueva clase de planificación AMP.

3.4. Algoritmos de planificación

El planificador *Completely Fair Scheduler* (CFS) fue desarrollado por Igno Molnar[1] e introducido en la versión del kernel 2.6.23. El parche que lo introdujo también modificó la arquitectura del planificador de Linux, con un diseño más modular. El nuevo diseño permite añadir con mayor facilidad nuevas clases de planificación al kernel y activarlas desde espacio de usuario asignándolas a procesos específicos.

El algoritmo CFS se basa en el uso de un árbol RB (*Red-Black*), ordenado por `vruntime`. Este algoritmo mantiene el siguiente proceso a ejecutar en el nodo de más a la izquierda del árbol. Usar un árbol RB para implementar la run queue de cada CPU permite reducir la complejidad de las distintas operaciones del planificador (al tener coste constante), algunas de las cuales se ejecutan cada *tick* de planificación. Asimismo la organización en árbol de la *run queue* permite al planificador mantener una representación temporal o *timeline* del orden en que los distintos procesos se ejecutarán.

Los distintos algoritmos implementados dentro del *framework de planificación* gestionan la *run queue* de cada CPU utilizando el algoritmo CFS, que no es consciente de la asimetría en la plataforma. Cada uno de los algoritmos del framework emplea una aproximación diferente para distribuir los ciclos de core *big* y core *small* disponibles en el sistema entre las distintas aplicaciones de la carga. A continuación se proporciona una descripción de los algoritmos más relevantes del *framework* que se han empleado en nuestro análisis experimental.

3.4.1. RR

La primera aproximación que se propuso para implementar un planificador que fuera consciente de la arquitectura AMP y mejorase la justicia fue una versión *asymmetry-aware* del conocido algoritmo de planificación **Round Robin**, que sencillamente intenta compartir de forma justa los cores big entre las aplicaciones [3]. Esta aproximación permite obtener mejor rendimiento global que los planificadores predeterminados en sistemas operativos de propósito general, que no son conscientes de la asimetría, y también reduce de forma significativa la variabilidad en los tiempos de ejecución de una aplicación

[1] Propuso el parche de unos 100 kB tras escribirlo en apenas 3 días a espaldas del resto de desarrolladores del kernel. Igno Molnar se basó algunas ideas de Con Kolivas y su Rotating Staircase Deadline Scheduler (RSDS), que no había sido aceptado para su inclusión en _mainline_.

registrados en distintas ejecuciones [18]. Por esta razón, RR se ha usado habitualmente como referente en estudios sobre planificación en plataformas AMP [3, 35]. Sin embargo, estudios recientes demuestran que este planificador constituye una solución ineficiente [28], ya que no considera los beneficios relativos que las aplicaciones experimentan al ejecutarse en un core big frente a uno small.

3.4.2. HSP

El algoritmo **HSP** (*High SPeedup*) [3, 15, 29] intenta optimizar el rendimiento del sistema, aumentando la prioridad de aquellas aplicaciones que muestren mayor rendimiento al ejecutarse en un core rápido frente a uno lento. Este planificador promueve un reparto injusto del procesador, ya que los cores rápidos son usados exclusivamente por las aplicaciones con mayor *speedup*.

3.4.3. EPI-Big

El planificador **EPI-Big** tiene como principal objetivo reducir el consumo energético. Con tal fin, asigna a los cores big los hilos con valores más bajos de consumo energético , definido como la energía por instrucción de las aplicaciones cuando se ejecutan en este tipo de core. Como se puede predecir, este planificador consigue reducir el consumo energético. No obstante, esto conlleva en ocasiones degradar sustancialmente el rendimiento y la justicia.

3.4.4. EEF-Driven

El planificador **EEF-Driven** [32] asigna los hilos a los cores big o small manteniendo la carga equilibrada en el AMP, y periódicamente reajusta las asignaciones teniendo en cuenta el EEF (*Energy Efficiency Factor*) de los diferentes hilos.

En la implementación de este algoritmo evaluada en [32], el planificador utiliza los contadores hardware en tiempo de ejecución para obtener un valor de EEF estimado para cada hilo. Esta implementación supone la creación de un modelo de estimación específico a cada arquitectura; para ello es preciso realizar un análisis preliminar exhaustivo de la plataforma [29]. En este TFG realizamos un estudio de una variante estática de este algoritmo que se alimenta con los EEFs medios de las aplicaciones de la carga de trabajo medidos previamente a la ejecución. A continuación procedemos a describir de forma más detallada el funcionamiento de la variante dinámica del algoritmo EEF-Driven, que podría ser implementada en base a los modelos de estimación que planteamos en el siguiente capítulo.

Cuando un hilo comienza su ejecución, el planificador EEF-Driven le asigna un valor por defecto de EEF[2], ya que no hay información previa de cada aplicación. La asignación inicial de los nuevos hilos se realiza de forma que se preserve el equilibrio de carga entre los cores. Tan pronto como un hilo comienza a ejecutarse, el planificador monitoriza el IPC y otras métricas relevantes de alto nivel requeridas para estimar el EEF del hilo. Cabe destacar que debido a diferencias microarquitectónicas entre los varios tipos de core, es posible que se necesiten diferentes conjuntos de eventos *Hardware* para estimar el EEF en cada core.

A medida que se ejecutan los hilos suceden dos cosas: (1) se da a conocer el EEF de los hilos que han comenzado a ejecutarse y (2) el EEF de hilos antiguos puede variar debido a su paso por diferentes fases de ejecución. El planificador debe asignar los hilos a distintos tipos de core en función de su EEF, por tanto se producen migraciones de hilos guiadas por eventos. En general, estas migraciones aseguran que el sistema cumpla estas dos reglas:

1. Todos los hilos de cores big tienen un mayor EEF que el máximo EEF de los hilos que corren en cores little.
2. La carga del sistema permanece equilibrada.

Para cumplir la primera regla, el planificador debe comprobar que el hilo de EEF mínimo de los cores big (T_{BC}) tiene un mayor valor que el EEF máximo de los hilos de cores small (T_{SC}). Esta regla puede incumplirse cuando cambia el EEF de un hilo o en el caso de que su estado cambie de *runnable* a *non_runnable* (p.ej., se bloquee por una operación de entrada/salida). En el primer escenario, el planificador hace cumplir la regla mediante un intercambio entre T_{BC} y T_{SC}. En el segundo caso, la migración de los hilos mencionados es suficiente para garantizar que las dos reglas se cumplen.

Para simplificar la toma de decisiones de planificación se mantiene una lista ordenada por EEF de los hilos en ejecución en cada core (solo hilos en estado *runnable*) , lo cual facilita la selección óptima de hilos a intercambiar: T_{BC} y T_{SC}. Por motivos de eficiencia, la lista del core big se mantiene en orden ascendente de EEF, mientras que la del core small tiene un orden descendente. De esta manera, encontrar los candidatos óptimos para intercambio en cualquiera de los escenarios mencionados tiene coste constante.

3.4.5. ACFS

La mayor parte de algoritmos de planificación que intentan optimizar el rendimiento o mejorar la eficiencia energética, como los ya descritos HSP [15, 29] o EEF-Driven [32], están sujetos a una importante limitación: se rigen por un factor fijo de mejora de estas propiedades (justicia, rendimiento o eficiencia energética). Al mismo tiempo, estos algoritmos de planificación son inherentemente injustos (como se expone en [32]). Esto

[2]Para este valor por defecto se utiliza el mínimo EEF observado en los _benchmarks_ SPEC CPU en la plataforma. De este modo, los hilos con un valor estimado relativamente bajo y los que han sido legítimamente asignados a los cores big no son relegados a los cores little cuando nuevos procesos entran en el sistema.

se debe al hecho de que optimizar el rendimiento o la eficiencia energética habitualmente conlleva la asignación de un subconjunto reducido de aplicaciones de la carga de trabajo a los cores big, relegando el resto a los cores small de menor rendimiento. Estas situaciones hacen difícil incorporar soporte de calidad de servicio en el algoritmo de planificación, ofrecer soporte de prioridades u ofrecer parámetros de configuración que permitan al administrador configurar el comportamiento del planificador.

El planificador *Asymmetry-Aware Completely Fair Scheduler* (**ACFS**) [31] se basa en el planificador CFS y lleva su planificación basada en justicia un paso más allá. Este algoritmo hace consciente al sistema operativo de la arquitectura asimétrica subyacente y permite ajustar el factor en el que se preserva la justicia a costa de degradar la eficiencia energética o el rendimiento global. Para conseguirlo, el algoritmo monitoriza el progreso de las aplicaciones e intenta mantener la justicia garantizando que todas ellas realizan un progreso similar. El hecho de monitorizar el progreso implica tener en cuenta los factores de ganancia de los distintos hilos de ejecución presentes en la carga de trabajo (*speedups*).

ACFS exporta un parámetro de configuración, llamado *Unfairness Factor*(UF), que permite al administrador del sistema incrementar el rendimiento global a costa de degradar la justicia. Para ello, el planificador incrementa gradualmente la prioridad de los hilos de la carga de trabajo que tienen un mayor factor de ganancia (*speedup*).

Recientemente se ha propuesto una variante de ACFS[32] que está equipada con un nuevo parámetro de configuración, llamado *EDP_Factor*, que permite dar mayor prioridad de forma gradual a la eficiencia energética en detrimento de la justicia. Esto se lleva a cabo incrementando la fracción de core big disponible para aquellas aplicaciones con un mayor Energy-Efficiency Factor (EEF).

Capítulo 4

Análisis experimental

En este capítulo se realiza un análisis de los resultados de la ejecución de *benchmarks* individuales y de cargas de trabajo multiprogramadas utilizando los algoritmos de planificación presentados en el Capítulo 3.

En nuestro estudio empleamos las versiones estáticas de los algoritmos HSP, EPI-big, EEF-Driven y ACFS. Estas variantes de los algoritmos no estiman en tiempo de ejecución los valores de EEF o SF de las aplicaciones sino que toman decisiones de planificación empleando los valores de EEF o SF medios observados durante la ejecución de cada aplicación sola en el sistema. El estudio de los algoritmos estáticos nos permite analizar de forma sencilla la efectividad de las distintas estrategias de planificación. No obstante, para permitir la implementación de versiones dinámicas de los algoritmos sobre la placa de desarrollo empleada (Odroid XU4) proporcionamos en este capítulo una serie de modelos de estimación basados en contadores hardware para estimar el SF y el EEF de un hilo en tiempo de ejecución.

El resto de este capítulo se estructura como sigue. La Sección 4.1 se analizan los resultados de caracterización de las aplicaciones. Estos resultados son necesarios para obtener el SF y el EEF de cada aplicación para alimentar a las variantes estáticas de los algoritmos *asymmetry-aware* considerados. En la Sección 4.2 se discuten los resultados obtenidos para los experimentos con cargas de trabajos multiprogramadas. Finalmente, en la Sección 4.3 se presentan los modelos de estimación para determinar el EEF y SF de las diferentes aplicaciones en tiempo de ejecución empleando contadores hardware.

4.1. Caracterización de aplicaciones

Los primeros experimentos se realizaron con el fin de caracterizar las aplicaciones SPEC CPU2000 y CPU2006. Para ello utilizamos la herramienta Het-Harness que permite sistematizar el lanzamiento de todos los *benchmarks*. Esta herramienta hace uso de PMCTrack para obtener información relevante de cada de las distintas aplicaciones mediante contadores hardware. Con los datos proporcionados por PMCTrack se calcularon las siguientes métricas:

- Instrucciones por ciclo.
- Accesos a la LLC (*last-level cache*) por cada mil instrucciones.
- Fallos de LLC por cada mil instrucciones.
- Energía en uJ.
- Energía neta en uJ (valor estimado).
- Energía neta por instrucción (*EPI*).

Estas métricas se obtuvieron para ambos tipos de core, y a partir de ellas se calcularon los factores SF y EFF que requieren los algoritmos de planificación estáticos usados en la siguiente sección.

En la Figura 4.1 se muestra el EEF y el SF de todas las aplicaciones de SPEC CPU. Como se puede apreciar, toda aplicación experimenta un mayor rendimiento al ejecutarse en un *core big* con respecto a ejecutarse en uno *core small*. Sin embargo, desde el punto de vista energético, la ejecución de una aplicación muy intensiva en memoria en un *core big* acarrea un gasto de energía innecesario. Este tipo de aplicaciones pasa gran parte del tiempo realizando accesos a memoria. Por lo tanto, ejecutar aplicaciones intensivas en memoria en un *core small* suele proporcionar un mayor rendimiento por vatio y permite destinar los cores *big* a la ejecución de aplicaciones más intensivas en CPU, que suelen extraer una mayor eficiencia energética de estos cores.

La Figura 4.2 muestra la energía neta por instrucción consumida en un *core big* (EPI_{big}) y la energía por instrucción consumida en uno *small* (EPI_{small}) frente al factor de ganancia o SF (*Speedup Factor*). Los resultados revelan que la mayor parte de aplicaciones consumen más energía al ejecutarse en un *core big* frente a uno *small*. Sin embargo, se observan algunas que a pesar de tener un *speedup* elevado, su consumo energético es muy pequeño, posiblemente debido a que hacen un uso más eficiente de las mejoras microarquitectónicas presentes en los *cores big*.

4.2. Cargas de trabajo

Los experimentos principales del proyecto tienen como objetivo analizar el comportamiento de diferentes algoritmos de planificación al ejecutar cargas de trabajo multiprogramadas compuestas por aplicaciones de los conjuntos de *benchmarks* SPEC CPU 2000 y CPU 2006. En la siguiente sección se detalla la metodología empleada para realizar los experimentos con cargas de trabajo multiprogramadas. En la Sección 4.2.2 se discuten los problemas de contención de recursos compartidos que se manifiestan al ejecutar algunas cargas de trabajo multiprogramadas. En la Sección 4.2.3 se discute la efectividad de los distintos algoritmos de planificación considerados para diversas cargas de trabajo multiprogramadas.

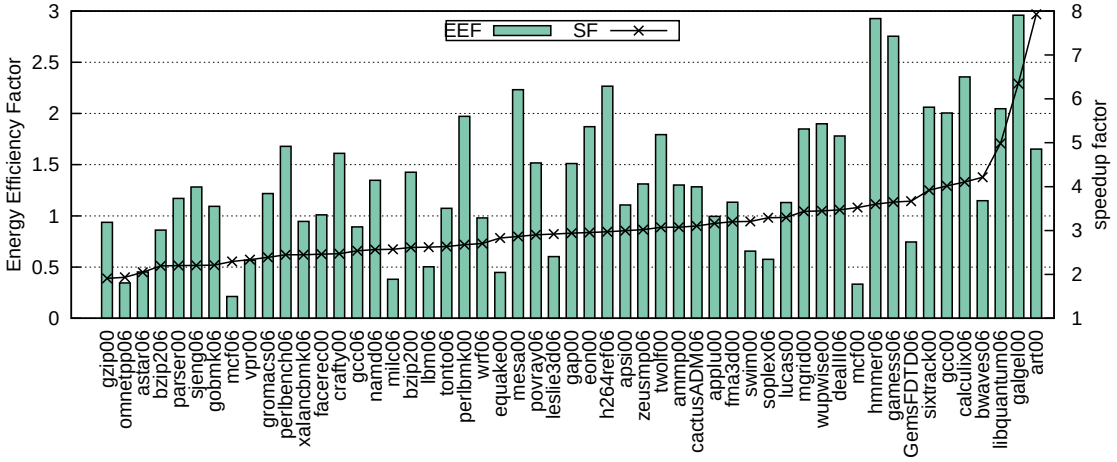

Figura 4.1: Factor de eficiencia energética y factor de ganacia observado para los benchmarks SPEC CPU cuando se ejecutan en cores big y small de la placa ODROID-XU4

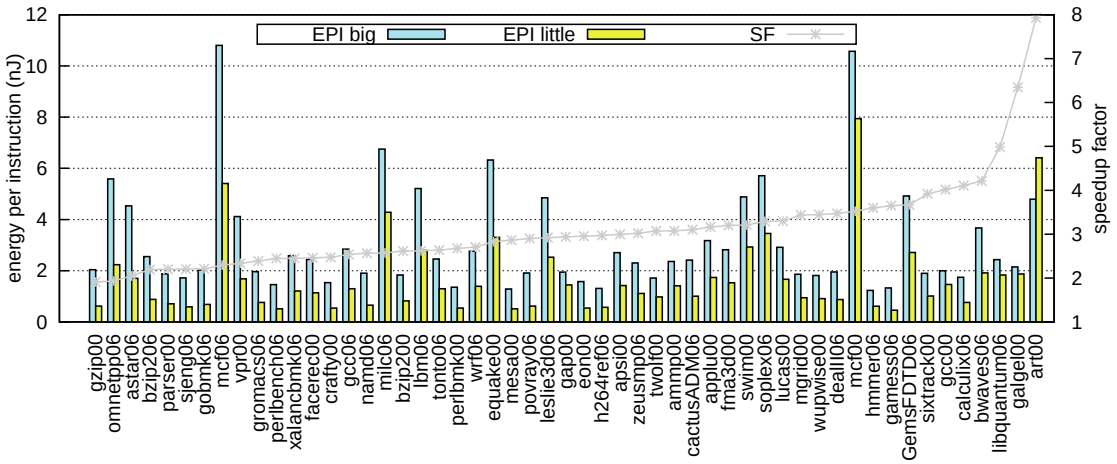

Figura 4.2: Factor de ganancia y energía por instrucción consumida en un core big y small observado para los benchmarks SPEC CPU cuando se ejecutan en cores big y small de la placa ODROID-XU4

Tabla 4.1: Cargas de trabajo.

Carga de Trabajo	Aplicaciones
W1	soplex06, equake00, perlbmk00, perlbench06
W2	povray06, equake00, perlbmk00, perlbench06
W3	mgrid00, swim00, h264ref06, perlbmk00
W4	sixtrack00, mcf00, mgrid00, h264ref06
W5	sixtrack00, mcf00, h264ref06, povray06
W6	art00, sixtrack00, gamess06, perlbmk00
W7	sixtrack00, equake00, perlbmk00, gobmk06
W8	sixtrack00, equake00, perlbmk00, gzip00
W9	mcf00, mgrid00, h264ref06, gobmk06
W10	art00, gamess06, hmmer06, povray06

Para mitigar los efectos de la contención por recursos compartidos en los experimentos, se ha utilizado una configuración asimétrica con dos cores big y dos small. De este modo solo hay 2 cores activos por cada cluster en la placa Odroid XU4 y se permite evaluar de forma mas representativa las diferencias que exhiben los algoritmos de planificación.

4.2.1. Metodología

La Tabla 4.1 muestra la composición de las cargas de trabajo multiprogramadas usadas en la evaluación de los algoritmos de planificación. Para construir estas cargas se empleó el siguiente procedimiento. En primer lugar, se construyeron múltiples cargas de 4 aplicaciones mediante la combinación de 15 programas SPEC CPU que cubren un amplio espectro de valores de SF y EEF. De las cargas generadas, se descartaron aquellas en las que los planificadores HSP y EEF-Driven realizan asignaciones de aplicaciones a cores big similares durante gran parte de la ejecución[1]. Las cargas de trabajo seleccionadas permiten ilustrar el potencial de cada uno de los algoritmos de planificación, en escenarios donde éstos realizan diferentes asignaciones de hilos a cores.

Una vez seleccionadas las cargas de trabajo, se utilizó la herramienta Het-Harness para lanzar secuencialmente cada una de estas cargas. Het-Harness puede configurarse para lanzar las cargas con un conjunto de algoritmos de planificación. Asimismo, el usuario puede indicar al sistema en ficheros de texto los valores de SF y EEF para cada aplicación, necesarios para hacer funcionar las variantes estáticas de los algoritmos de planificación considerados. Estos valores fueron obtenidos durante los experimentos de caracterización de aplicaciones (Sección 4.1).

Al lanzar una carga de trabajo, se asegura que todas las aplicaciones de la carga se inician simultáneamente. Cuando una aplicación termina, el sistema lanza una nueva instancia de la misma hasta que la aplicación con el tiempo de finalización mayor se ejecuta tres veces. De esta forma, se mantiene la carga de trabajo continua, y se eliminan los problemas derivados de que las aplicaciones tengan distintos tiempos de finalización.

[1]Para estas cargas,las aplicaciones con valores altos de SF resultan ser también las que tienen valores altos de EEF

4.2.2. Contención de recursos compartidos

Ninguno de los algoritmos estudiados en este TFG tienen en cuenta los problemas de contención de recursos compartidos. Este efecto se hizo evidente durante un estudio preliminar realizado con cargas de trabajo formadas por múltiples aplicaciones intensivas en memoria (como `equake` o `soplex`). En este escenario, las aplicaciones intensivas en memoria ocasionan contención en la cache compartida y degradan el rendimiento del resto de aplicaciones de la cargas de trabajo. Además, la degradación del rendimiento al compartir cache no es uniforme entre aplicaciones ya que depende de la cantidad de espacio en cache que necesitan para ejecutarse sin que el número de fallos de cache aumente significativamente.

Cabe destacar que algunos algoritmos de planificación (especialmente HSP) asignan las aplicaciones intensivas en memoria a un mismo core por períodos mayores de tiempo que otros planificadores (como ACFS o EEF-Driven). Por lo tanto, no todos los algoritmos están sujetos al mismo grado de degradación del rendimiento por contención. También observamos que la contención en las caches compartidas de nivel 2 en los cores *small* no es tan alta, provocando una mínima degradación del rendimiento en las cargas de trabajo exploradas. Esta situación puede explicarse por el hecho de que los cores *small* con ejecución en orden no son capaces de resolver múltiples fallos de cache [2], lo que provoca un ratio inferior de accesos a la cache de último nivel y una utilización inferior del ancho de banda del bus. En conclusión, se reduce del grado de contención de recursos compartidos en este tipo de cores.

Para asegurar una comparación justa de los diferentes algoritmos y mitigar el impacto de la compartición de recursos, se decidió aplicar particionado de cache en las cargas de trabajo con múltiples aplicaciones intensivas o sensibles a compartir cache. Específicamente, se dividió la cache compartida de forma que cada aplicación intensiva en memoria solo pudiese acceder a la mitad de la cache. Al mismo tiempo, se permitió al resto de aplicaciones usar el espacio de la cache bajo demanda (sin particionado de cache). Ya que la plataforma Odroid XU4 no tiene soporte hardware para particionar la cache, se usó la herramienta PALLOC explicada en la sección 2.2.2. Esta configuración nos permitió emular de forma aproximada un escenario libre de contención, que constituye un mejor entorno para evaluar el potencial que se deriva de usar planificadores conscientes de las arquitecturas asimétricas.

4.2.3. Discusión

Las figuras 4.3, 4.4 y 4.5 muestran los valores del *Energy-Delay Product* (EDP), el *Aggregate Speedup* (ASP) y la injusticia (*Unfairness*) de las cargas de trabajo W1-W10. Los valores presentados están normalizados con respecto al algoritmo de planificación HSP. Se puede apreciar una tendencia general que nos indica que optimizar una métrica (como el EDP) puede conllevar una degradación sustancial de otra (como el ASP).

[2] Un fallo de cache ocasiona una parada del pipeline en cores con ejecución en orden. En cambio, en un core con ejecución fuera de orden un fallo de cache no impide que el core pueda continuar lanzando instrucciones.

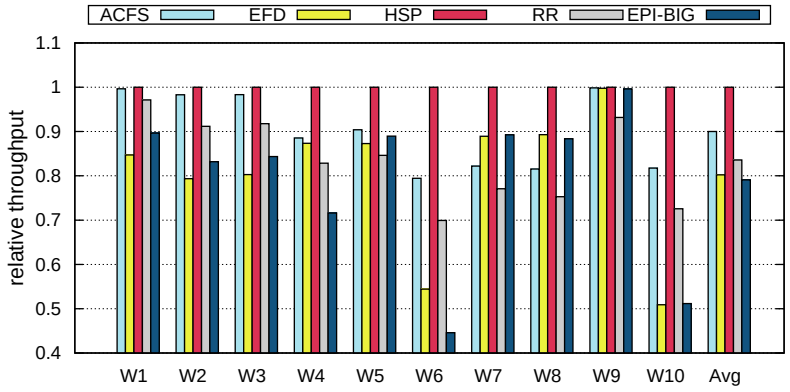

Figura 4.3: Rendimiento relativo para las cargas de trabajo W1-W10

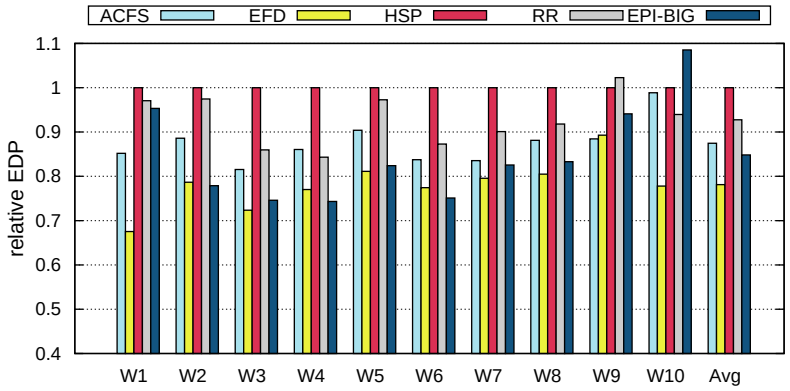

Figura 4.4: EDP relativo para las cargas de trabajo W1-W10

Los resultados procedentes de los algoritmos que buscan optimizar el rendimiento indican que el algoritmo de planificación HSP consigue los mejores valores de ASP a costa de degradar sustancialmente el EDP. En contraposición, los algoritmos de planificación EEF-Driven y EPI-Big obtienen los mejores resultados de EDP (cuanto más bajos, mejor) a costa de degradar significativamente el rendimiento. En los resultados se observa que HSP experimenta una degradación del EDP (peor eficiencia energética) de un 22 % respecto a EEF-Driven, y un 16 % relativo a EPI-Big; a cambio de mejorar el rendimiento un 21 % y 23 %, respectivamente.

Al centrar la atención en los algoritmos que dan más prioridad a la eficiencia energética (EEF-Driven y EPI-Big), se observa que el algoritmo EEF-Driven es el que genera un mayor ahorro energético. Se obtienen valores de EDP solo un 6 % inferiores a los de EPI-Big. Por el contrario, podemos encontrar algunas cargas de trabajo como la W6, donde el EDP es apenas un 3 % más bajo para el algoritmo EPI-Big. La mejora generalizada que ofrece EEF-Driven se debe a que este algoritmo de planificación considera tanto el factor de ganancia (SF) como el ratio de energía consumida por instrucción en el core big (EPI).

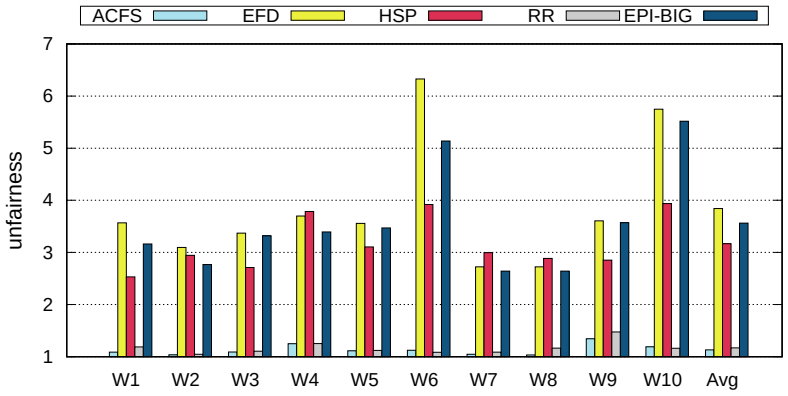

Figura 4.5: Injusticia para las cargas de trabajo W1-W10

Al analizar los resultados de los algoritmos de planificación centrados en promover la justicia, se puede apreciar que ACFS y RR presentan un comportamiento cercano al óptimo (en cuanto a justicia) y casi idéntico (ACFS es un 1 % mejor que RR). Como se puede observar en los resultados, los algoritmos RR y ACFS son los únicos conscientes de la justicia. El resto de algoritmos degradan esta métrica al intentar optimizar otras propiedades, ya sea el EDP o ASP. Por ejemplo, el algoritmo HSP y EEF-Driven presentan una degradación de la justicia de un 55 % y 70 % respectivamente con respecto al algoritmo ACFS. Es necesario subrayar que el algoritmo ACFS presenta un rendimiento sólo un 10 % inferior en promedio al del algoritmo HSP. De esta forma, ACFS es un algoritmo de planificación que maximiza la justicia y al mismo tiempo mantiene un rendimiento cercano al máximo observado (el de HSP). Esto se deriva principalmente del hecho de ACFS proporciona una fracción de tiempo de core *big* mayor a las aplicaciones que hacen un uso más eficiente de ese tipo de cores. Además, con ACFS el período de tiempo que se ejecutan simultáneamente aplicaciones intensivas en memoria junto con aplicaciones sensibles a compartir cache se reduce, mitigando el efecto de la contención de recursos compartidos.

4.2.4. Efectividad de los parámetros de configuración del planificador ACFS

Una de las principales desventajas de la mayoría de algoritmos de planificación analizados en la sección anterior es que se rigen por un factor fijo de mejora de las diferentes propiedades que intentan optimizar, a cambio de degradar la justicia u otras métricas. Por el contrario, el algoritmo ACFS muestra un comportamiento más flexible. Este algoritmo está provisto de dos parámetros configurables (*EDP_factor* y *Unfairness_Factor*) que permiten al administrador del sistema la posibilidad de reducir la justicia en el sistema de forma incremental a cambio de mejorar el rendimiento o la eficiencia energética. Cuando estos parámetros tienen su valor por defecto, ACFS presenta el comportamiento

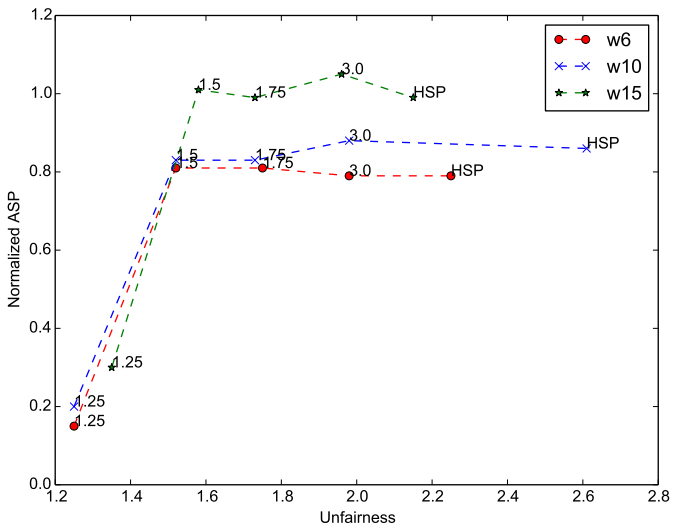

Figura 4.6: Injusticia vs rendimiento normalizado para diferentes valores de Unfairness Factor del algoritmo ACFS

observado en los experimentos de la sección anterior. En otras palabras, el algoritmo intenta alcanzar el máximo rendimiento (ASP) reduciendo al mínimo el valor de la injusticia. La figura 4.7 muestra como la variación del factor *EDP_factor* afecta a la justicia y consumo energético para tres cargas de trabajo seleccionadas.

Los resultados muestran como se comporta el algoritmo ACFS para diferentes valores de *EDP_factor*, para el valor por defecto tenemos el valor más bajo de injusticia. Pero al incrementar el factor *EDP_factor* para favorecer el EDP observamos que la energía consumida baja hasta saturar en unos valores próximos a los del algoritmo EEF-Driven, mientras que la justicia se degrada.

En la Figura 4.6 se representa la mejora de rendimiento que se deriva de aumentar el parámetro *Unfairness_ Factor* en tres cargas de trabajo seleccionadas. Al incrementar los valores de este factor (por ejemplo de 1.25 a 1.5), se produce una mejora en el rendimiento de las tres cargas de trabajo y una degradación de la justicia. Se puede observar que llega un punto en que la mejora de rendimiento satura, en unos valores muy próximos a los del algoritmo HSP.

En conclusión, se ACFS constituye un algoritmo de planificación capaz de adaptarse a las necesidades del administrador y los requisitos del sistema. El usuario es capaz de favorecer el ahorro energético, el rendimiento o la justicia en mayor o menor medida. La problemática que se genera es la misma que la de cualquier otro algoritmo, que al favorecer una propiedad se degrada otra, pero es el usuario el que decide la interrelación estas propiedades.

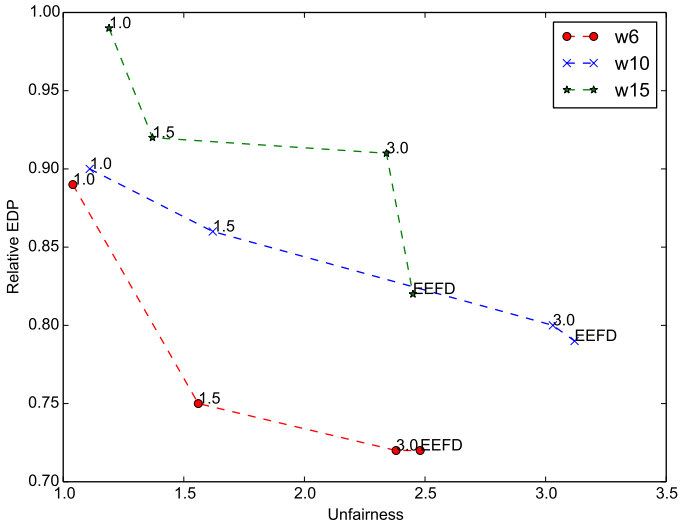

Figura 4.7: Injusticia vs EDP relativo para diferentes valores de EDP_Factor del algoritmo ACFS

4.3. Modelos de estimación

En la sección anterior se procedió a discutir la efectividad de las variantes *estáticas* de los algoritmos de planificación HSP, ACFS y EEF-Driven. Estas variantes de los algoritmos resultan subóptimas dado que no tienen en cuenta que una aplicación puede atravesar múltiples fases con distinto SF [16] o EFF [32] a lo largo de su ejecucion.

Las variantes *dinámicas* de estos algoritmos permiten solventar este problema. En este escenario, el planificador ha de determinar el SF y/o EFF de un hilo en tiempo de ejecución. Si bien la medida directa del SF de un hilo en ejecución es posible e ineficiente (midiendo el IPC de la aplicación ejecutándose en ambos tipos de core), estudios recientes han demostrado que esta aproximación da lugar a grandes imprecisiones en los valores obtenidos debido a la existencia de distintas fases de ejecución en las aplicaciones [36, 31]. La medida directa del EFF de un hilo mientras se ejecuta como parte de una carga de trabajo multiprogramada no es posible en la mayor parte de sistemas [32]. Esto se debe a que los sistemas actuales no implementan sensores o registros que proporcionen información aislada sobre la contribución al consumo energético total de un hilo individual de la carga. En la práctica, esto hace que sea muy complicado medir el ratio de energía consumida (de memoria y core) por instrucción de hilos específicos, lo cual es necesario para el cálculo del EEF.

Para aproximar el SF de un hilo en tiempo de ejecución pueden emplearse heurísticas específicas de arquitectura [15] o modelos de estimación [15, 29, 32]; ambas aproximaciones están basadas en el uso de contadores hardware. Se ha demostrado recientemente que los modelos de estimación también resultan eficaces para aproximar el EEF [32].

Para complementar nuestro estudio experimental y hacer posible una futura implementación de las estrategias dinámicas de los distintos algoritmos, hemos construido una serie de modelos de estimación basados en contadores hardware para predecir el SF y el EEF en la placa Odroid XU4. Nótese que para realizar esta predicción, son necesarios dos modelos de estimación: uno para predecir el valor desde el core *big* (a partir de eventos hardware de ese core) y otro para obtener una predicción desde el core *small*. Esto se debe a que los modelos de estimación de SF o EEF, dependen de métricas de rendimiento, como el IPC o la tasa de fallos de cache, y el valor de estas métricas puede variar entre tipos de core. Al emplear modelos basados en regresión aditiva, los coeficientes de regresión y las métricas relevantes para predecir el SF/EEF desde pueden ser diferentes en los modelos construidos para la predicción desde cada tipo de core.

Para construir los modelos de estimación de EEF y SF hemos empleado la metodología que se describe en [29]. En resumen, esta metodología conlleva la realización de los siguientes pasos:

1. Seleccionar un conjunto AP de aplicaciones secuenciales representativas y un conjunto M de métricas de rendimiento.
2. Ejecutar las aplicaciones de AP en ambos tipos de core para obtener su SF medio y el valor de las métricas de rendimiento en M usando los contadores hardware del procesador. Para ello puede emplearse la herramienta PMCTrack [26].
3. Los resultados del paso anterior son procesados con la herramienta WEKA [10]. Esta herramienta permite aplicar numerosas técnicas de minería de datos y aprendizaje automático para inferir relaciones entre un conjunto de observaciones (o atributos de entrada) y una variable objetivo. En el problema de estimación de SFs (o EEFs), las observaciones se corresponden con distintas métricas de rendimiento obtenidas en un core concreto y la variable objetivo es el SF (o el EEF). Empleando el motor de regresión aditiva de WEKA, se generan dos modelos de estimación: uno para poder obtener una predicción desde el core *big* y otro desde el core *small*. Es importante destacar que, al generar los modelos, WEKA descarta automáticamente aquellas métricas de M menos relevantes para el proceso de estimación. Por lo tanto los modelos resultantes pueden no depender de todas las métricas en M.

Las figuras 4.8 y 4.9 muestran una comparativa entre los valores observados (reales) y los valores estimados en ambos tipos de cores para el EEF y el SF respectivamente. Para obtener esos modelos empleamos información de monitorización hardware recabada para los *benchmarks* SPEC CPU2000 y CPU2006 en la placa Odroid XU4. La monitorización de la ejecución completa de todos los *benchmarks* es muy costosa en tiempo, por lo que se usó la información de un número significativo de ventanas de instrucciones. En el caso de los modelos de predicción para el EEF, los coeficientes de correlación asociados a la estimación en los cores big y small fueron 0.96 y 0.92, respectivamente. Estos coeficientes de correlación obtenidos nos indican que el modelo propuesto proporciona una buena

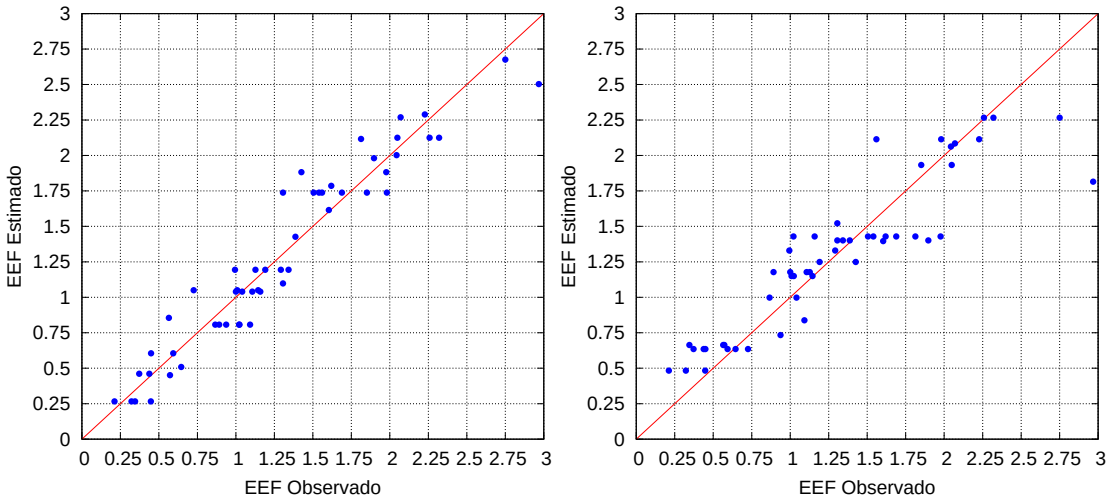

Figura 4.8: Predicción de EEF en cores big (izquierda) y small (derecha) mediante regresión aditiva.

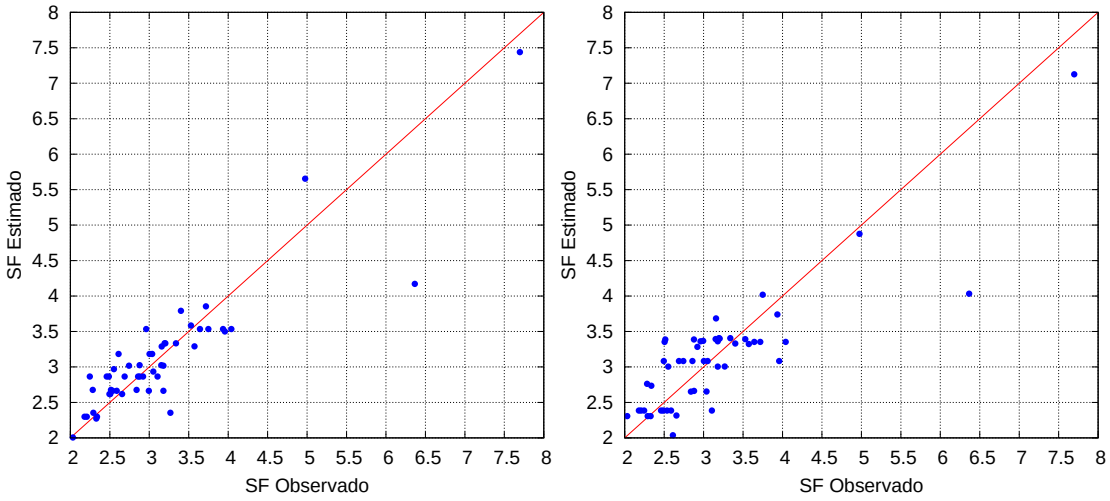

Figura 4.9: Predicción de SF en cores big (izquierda) y small (derecha) mediante regresión aditiva.

Tabla 4.2: Métricas de rendimiento y eventos *Hardware* asociados para hacer predicciones dinámicas en ambos tipos de core.

Eventos hardware	**Métricas**
Instrucciones retiradas,	Instrucciones,
Instrucciones por ciclo,	Ciclos,
Fallos de LLC por 1K instr.,	Fallos de cache LLC,
Accesos a LLC por 1K instr.,	Accesos a LLC,
Fallos de cache L1 por 1K instr.,	Fallos de cache L1,
Accesos a cache L1 por 1K instr.,	Accesos a cache L1,
Accesos a memoria de datos por 1K instr.,	Accesos a memoria de datos,
Fallos de ITLB por 1M instr.,	Fallos de ITLB,
Fallos de DTLB por 1M instr.,	Fallos de DTLB,
Fallos de predicción de saltos por 1K instr.	Fallos de predicción de saltos

estimación del EEF a través de métricas de alto nivel que el sistema operativo puede monitorizar fácilmente en tiempo de ejecución. Los coeficientes de correlación observados para las predicciones de SF son inferiores (0.90 y 0.86 para los cores big y small, respectivamente), lo cual revela que la calidad de la estimación es ligeramente peor.

Empleando el modelo de SF, sería posible implementar una versión dinámica de los algoritmos ACFS y HSP sobre la placa Odroid XU4. Análogamente, la implementación de una variante dinámica de EEF-Driven en esta plataforma podría llevarse a cabo empleando el modelo de EEF. Para ello, el planificador debería monitorizar los eventos hardware y las métricas de alto nivel de las que dependen los modelos generados, que se muestran en la tabla 4.2.

Capítulo 5

Conclusiones

5.1. Conclusiones

Hoy en día, los dispositivos móviles demandan cada vez cantidades inferiores de energía para mejorar la autonomía y el rendimiento por vatio, los procesadores asimétricos representan una importante alternativa a los *chips* simétricos convencionales. Los AMP ofrecen la posibilidad de ejecutar diferentes tipos de aplicaciones adaptándose a las necesidades de procesamiento de cada una.

La mayor parte de planificadores de sistemas operativos de propósito general no son conscientes de las arquitecturas asimétricas subyacentes. Un número creciente de fabricantes de hardware están incorporando esta arquitectura, principalmente en sus procesadores móviles como la familia big.LITTLE de ARM. A pesar de ello, los sistemas operativos actuales no están plenamente adaptados a este tipo de arquitecturas, y por ello, no explotan al máximo las capacidades que ofrecen los AMPs.

En este Trabajo de Fin de Grado se ha llevado a cabo un estudio experimental sobre el impacto derivado de optimizar diferentes tipos de métricas en la placa Odroid XU-4, que integra un procesador ARM big.LITTLE. En concreto, se ha hecho especial énfasis en estudiar la interrelación que existe entre el rendimiento, la justicia y la eficiencia energética. Para realizar nuestro análisis, se han evaluado implementaciones en el kernel Linux de una serie algoritmos de planificación capaces de optimizar una o varias métricas. Muchos de estos algoritmos toman decisiones de planificación considerando el beneficio relativo en rendimiento y/o eficiencia energética que las distintas aplicaciones derivan de usar los cores de alto rendimiento frente a los de bajo consumo.

Nuestro estudio revela que la optimización de distintas métricas (p.ej., rendimiento global y justicia) representan con frecuencia objetivos contrapuestos. Concretamente, se ha demostrado que se produce una degradación sustancial de la justicia al optimizar rendimiento global o eficiencia energética. En este ámbito, consideramos que el planificador ACFS y sus variantes constituyen una estrategia de planificación que destaca sobre las demás. Esencialmente, la mayor parte de algoritmos evaluados ofrecen una relación fija

de optimización y degradación entre las diferentes métricas expuestas. Por el contrario, ACFS está dotado de una serie de parámetros de configuración que permiten al administrador del sistema configurar el grado en el que se mantiene la justicia en el sistema a costa de degradar rendimiento o eficiencia energética.

En el ámbito de la optimización de la eficiencia energética, evaluamos el comportamiento de dos planificadores. El primero, al que llamamos EPI-big, asigna a cores de alto rendimiento aquellas aplicaciones que exhiben un menor ratio de energía por instrucción (*EPI*). Nótese que este tipo de core, complejo desde el punto de vista de la microarquitectura, exhibe un consumo de energía mucho más elevado que los cores simples de la plataforma; por lo tanto reducir el consumo energético del conjunto de cores de alto rendimiento es crucial para mejorar la eficiencia energética del sistema asimétrico. El segundo algoritmo, llamado EEF-Driven, asigna de forma preferente a los cores de alto rendimiento las aplicaciones que exhiben un mayor factor de ganancia (rendimiento relativo al core de bajo consumo) y a la vez un menor ratio de energía por instrucción (*EPI*) en el core de alto rendimiento. Nuestro análisis revela importantes conclusiones:

- Optimizar el *energy delay product* (EDP) en procesadores multicore asimétricos puede conllevar una degradación sustancial del rendimiento.
- En la mayoría de casos favorecer la justicia conlleva sacrificar tanto rendimiento como eficiencia energética.
- Entre todos los algoritmos considerados, la estrategia EEF-Driven consigue el mejor EDP (mínimo) para todas las cargas de trabajo, y a la vez ofrece mayor rendimiento que la estrategia EPI-big. Este hecho pone de manifiesto que se pueden conseguir reducciones sustanciales en el EDP considerando tanto los factores de ganancia de las aplicaciones como el consumo energético de las mismas en el core de alto rendimiento.

5.2. Valoración del TFG

El desarrollo de este Trabajo de Fin de Grado ha requerido la utilización de un conjunto herramientas de gran complejidad, que han sido desarrolladas por miembros del grupo de investigación ArTeCS y con la participación de estudiantes de distintos Trabajos de Fin de Grado en los últimos años. Éste ha sido un Trabajo de Fin de Grado muy interesante y con un gran apartado teórico, que ha permitido obtener una serie de conocimientos y aptitudes típicas en labores de investigación de nivel profesional. Consideramos que los resultados obtenidos han sido positivos y han reforzado la hipótesis, ya comprobada por otros autores [17, 15, 29, 31, 32], de que los sistemas asimétricos representan una alternativa de futuro para la computación de altas prestaciones (HPC) y en el marco de los dispositivos móviles.

5.3. Trabajo futuro

La principal línea de trabajo que podría desarrollarse a partir de los resultados obtenidos en este Trabajo de Fin de Grado sería la implementación de las versiones dinámicas de los algoritmos de planificación explorados. En nuestro estudio se han analizado únicamente las versiones estáticas de estas estrategias, que no reaccionan a las distintas fases de ejecución que exhiben algunas aplicaciones. Para construir los algoritmos dinámicos (conscientes de las distintas fases de ejecución), es preciso dotar al planificador de un mecanismo para determinar en tiempo de ejecución los factores de ganancia y los ratios de eficiencia energética de cada hilo en el sistema. Esto puede lograrse implementando los distintos modelos de estimación propuestos en la Sección 4.3, que están basados en contadores hardware. Estos modelos se pueden implementar de forma sencilla mediante un módulo de monitorización de PMCTrack [26], que alimentase a los planificadores con estimaciones de los factores de ganancia y los ratios de eficiencia energética de cada hilo. Siguiendo esta aproximación, la implementación del planificador en el *kernel* del sistema operativo estaría completamente desacoplada de la arquitectura subyacente. Lo que es más importante, esto haría posible mantener la implementación base de la versión estática del algoritmo, que ahora sería alimentada por estimaciones de los distintos parámetros (por hilo) necesarios para tomar decisiones de planificación a lo largo del tiempo.

Otro aspecto clave que no ha sido suficientemente estudiado en el contexto de los AMPs es el efecto de la contención de recursos compartidos, como es la memoria cache o el bus. En nuestro estudio experimental aplicamos particionado de cache para aislar aplicaciones intensivas en memoria en una región específica de cache, para evitar así que pudieran perjudicar a otras aplicaciones que sean muy sensibles a compartir cache. A pesar de que en el contexto de sistemas multicore simétricos se han propuesto muchos algoritmos que son capaces de lidiar con la contención por recursos compartidos [41], no existen aún propuestas relevantes que analicen a fondo este problema en multicore asimétricos. Por lo tanto, creemos que combinar estrategias para mitigar los efectos de la contención cache con algoritmos de planificación conscientes de la asimetría podría suponer una interesante vía de investigación.

Un área interesante en cuanto a trabajo futuro es la implementación y evaluación de los de algoritmos de planificación considerados en este TFG, pero sobre una plataforma móvil asimétrica real, por ejemplo usando el sistema operativo Android (basado en el kernel Linux). Esto permitiría evaluar las potenciales mejoras de usar algoritmos de planificación conscientes de la asimetría ejecutando cargas de trabajo formadas por distintas aplicaciones móviles. Así mismo, se podrían construir *benchmarks* específicos para comprobar el impacto de las mejoras proporcionadas por los distintos algoritmos en la duración de la batería. Esto supondría un aliciente para que más fabricantes opten por este tipo de procesadores y se impulse el desarrollo de un mejor soporte para sistemas AMPs en sistemas operativos de propósito general.

Apéndice A

Introduction

Technological evolution of electronic components has followed an exponential growth rate in the past few decades. This has produced a transition from gigantic mainframes to powerful mobile devices that are currently available, they bring every imaginable kind of software tool into our pockets. Since its presentation in 1965, the renowned Law of Moore has stayed valid as commandment of computers. However, the duty of its fulfilment has forced chip manufacturers to maintain a technological growth based in the increment of processor frequency. This strategy has pushed out other techniques to improve throughput that are being rediscovered nowadays, such as old vectorial architectures like the Cray-1, example of a SIMD architecture (Single Instruction Multiple Data).

The paradigm in microprocessor's design has evolved a great deal in the recent years. The rapid ascension in energy consumption and thermal dissipation issues had made obvious that frequency scaling as a technique of improving throughput could not be maintained as an improvement strategy. From that moment on, manufacturers opted to include multiple cores in the same chip, which has been possible due to the size reduction of transistors and other components. Besides, other architecture improvements have allowed to achieve a better use of thread level parallelism, by means of including multiple functional units in the same core (simultaneous multithreading).

Conventional asymmetric multicore processors, which are formed by identical cores, are present in a huge number of electronic devices. They could be classified into two main types:

On the one hand, high throughput processors. They integrate high power-consuming cores that function at high frequencies, such as the Intel Core i7-6700K. These kind of cores try to optimize performance by using complex techniques such as out-of-order execution or instruction multi-issuing, producing an increment in energy consumption as a trade-off.

On the other hand, processors constituted by low power-consuming cores that function at lower frequencies. Their main goal is energy saving, such as the Intel Atom or the ARM Cortex A7. These are processors mostly used in mobile devices in which battery life is a fundamental aspect. It is remarkable how these processors are able to offer high throughput to applications that showcase an elevated thread level parallelism, given the fact that pipeline stalls that block the execution in one core may not interrupt the execution in other cores.

The mentioned examples only have a limited number of cores, commonly between two and four. But this processor schemes are applied to bigger computing requirements. First of all, we can think of high performance computing in the form of servers and supercomputers, a common model in this context is the Intel Xeon family, whose last chip (E7-8870) counts with 18 physical cores. The second kind of processors has a wider presence in our daily life, the Graphic Processors Units (GPUs). For example, the new NVIDIA GTX 1080 is able to handle up to 2560 CUDA cores. In spite of its specific design for rendering graphics, they have spread as an alternative to conventional processors in fields like data mining, artificial intelligence or HPC. This stems from the fact that they provide the ability to execute in parallel a great number of simple tasks over huge amounts of data.

It is worth highlighting that current applications show off an enormous variety of computing requirements. There are some applications that stay the majority of time executing arithmetic instructions (CPU intensive), and they usually exploit more efficiently the principles of locality (in time and space) while accessing the cache memory. For this applications, the throughput is not substantially affected by the features of memory hierarchy. On the contrary, there are other applications (known as memory intensive) which expend most of the time with the pipeline stalled due to cache misses waiting to be resolved.

The existence of applications with divergent computing needs provoke that conventional multicore processors result in a suboptimal alternative in certain situations. For example, executing a memory intensive task in a high performance core does not allow to exploit the full potential of the microarchitecture features offered in this cores, designed to optimize the instruction level parallelism. Nonetheless, running a CPU intensive program in a low energy-consuming core is not the best alternative from the energy efficiency point of view, even if the core requires less power, the poor throughput achieved by this small cores could lead to an increment in the overall energy consumed.

The diversification of computing requirements has impulsed the beginning of a new stage in hardware design, in which the chips are built with components specialized in certain kinds of computing problems. In this context the asymmetric multicore processors[16, 17] were proposed as an alternative with better support than conventional processors to deal with heterogeneous workloads.

The rest of this chapter is structured as follows. In the Section A.1 it is introduced the concept of asymmetric multicore processor. The Section A.2 illustrates the motivation of this work. In Sections A.3 and A.4 are presented the goals of this project and the work plan. Finally, the Section A.5 describes the structure of this report.

A.1. Asymmetric Multicore Processors

An asymmetric multicore processor (AMP) integrates cores with different features in the same chip. On the one hand, there is a group of fast cores, with high throughput and energy consumption, which run at high frequencies and implement complex microarchitecture techniques such as out-of-order execution or multiple instruction issuing. On the other hand, this processors include a group of slower but simpler cores. The small cores operate at lower frequencies and make use of an ordered pipeline, resulting in lower energy consumption. Due to its complexity, high throughput cores often occupy a bigger on the chip than the smaller ones [17, 12]. Along this project, we refer to high throughput cores as fast or big and low energy-consumption cores as slow or small.

Asymmetric multicore processors provide higher flexibility than its symmetric counterparts, given the fact that they behave as a high throughput core for CPU intensive applications, without sacrificing energy efficiency. It is remarkable that memory intensive applications, which usually raise more pipeline stalls, can be executed in a small core without a significant throughput decrease but providing better energy efficiency to the system.

A popular example of commercial AMP is the big.LITTLE from ARM [2], which is included in many commercial off-the-shelf mobile devices. Intel has showed interest in this kind of architectures and the QuickIA prototype [4] is a clear example. Current asymmetric multicore processors present a single-ISA (instruction set architecture), which entails an easier development process [25]. This approach contrasts with the one followed by other heterogeneous systems like the IBM Cell [9], where different cores expose different ISAs.

Energy efficiency represents a key aspect in the high performance computing environment, where one of the major costs is the electric bill. AMP architectures could be very useful in this environment combined with parallel programming techniques such as the OpenMP framework. This would allow programmers to control the mapping of certain applications (or even specific execution phases) to a type of core, resulting in a more efficient use of parallelism. Firstly, fast cores could be used to execute sequential and CPU intensive phases. Secondly, slow and efficient cores would handle multithreaded phases. This strategy could boost significantly the system energy efficiency [1]. As a matter of fact, it is possible to improve global throughput if you dedicate big cores to run CPU intensive program phases and slow cores to execute memory intensive phases [17, 36].

A.2. Motivation

Previous work has demonstrated that in order to exploit the asymmetric multicore processors to their full potential, the operating system must notice the relative benefit (*speedup*) that an application experiences when running in a fast core relative to running in a slow one[17, 15, 31]. Some applications make a better use of microarchitecture features present in high throughput cores; these applications usually experience a significant

speedup from running in a fast core relative to running in a slower one. On the contrary, other applications, who raise frequent pipeline stalls, are not able to benefit as much from running in a fast core. The relative speedup of each application must be taken into account when making scheduling decisions in order to improve three essential properties: productivity or global throughput, energy efficiency and justice. The Chapter 3 a set of metrics to quantify the effectiveness of different scheduling algorithms in optimizing this properties.

State-of-the-art schedulers from general purpose operating systems are not aware by default of the speedup diversity in a multi-application workload. Recently, a few extensions were added to the Linux kernel to provide a better support for scheduling in asymmetric multicore processors running in an interactive environment. Nowadays. these extensions are mainly used in mobile devices running in Android over the Linux kernel. It is worth to remark that this extensions, known as Heterogeneous Multi-Processing (HMP) are not integrated in the mainline kernel version. The extensions can be obtained by applying the patch `SCHED_HMP` [24], that introduces the following functionality in the scheduler. The operating system classifies threads into two categories: *light* and *heavy*. Firstly, *light* tasks are those who spend most of the time blocked by I/O operations, for example activities who deal with user interactions. Lastly, *heavy* tasks spend the majority of time executing arithmetic instructions in the CPU. In this context, the scheduler assigns preferentially *heavy* tasks to big and high throughput cores, and relegates *light* tasks to small and energy efficient cores.

Even if the `SCHED_HMP` patch improves energy efficiency in interactive environments, the scheduler extension present two important limitations. Firs of all, when it classifies tasks in *light* or *heavy*, does not realize the relative speedup that each application experiments when running in a high throughput core vs a low consuming one. In particular, all CPU intensive tasks are classified as *heavy*, no matter what value of speedup is derived from running in a fast core or the degree of use of the memory hierarchy. As a consequence, the tasks are mapped in a suboptimal way to their running cores from the energy efficiency or throughput point of view. Secondly, when the number of *heavy* tasks is greater than the number of fast cores, some of them are relegated to slow cores. Moreover, the scheduler does not take any measure to guarantee that all tasks of this kind fair-share the fast cores in a way that they make a similar progress in the AMP. This situation constitutes an important limitation in the context of high performance computing, where most of the applications in the system load are CPU intensive (*heavy tasks*).

In the context of multi-applications workloads, the `SCHED_HMP` limitations cause a lot of variability in completion times across executions of the same program. To illustrate this issue, we proceeded to make the next experiment with a workload including 4 SPEC CPU benchmarks (`galgel`,`gamess`,`hmmer`,`povray`). We run the workload 8 times in an AMP with 2 high throughput cores (Cortex A15) and 2 low power-consuming cores (Cortex A7). A different launch order was followed in each execution and the workload was kept continuous during 20 minutes (the applications were relaunched if necessary). The Figure A.1 shows the mean completion times of the applications for each execution.

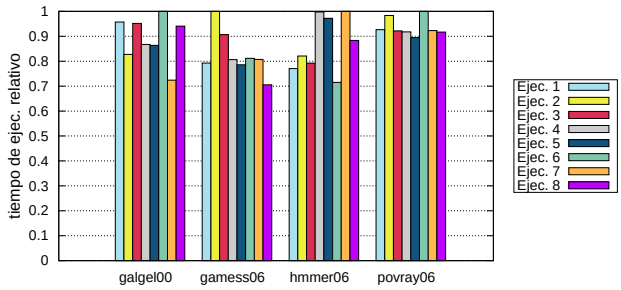

Figure A.1: Normalized completion times of the different applications of the workload permutations running under SCHED_HMP scheduler.

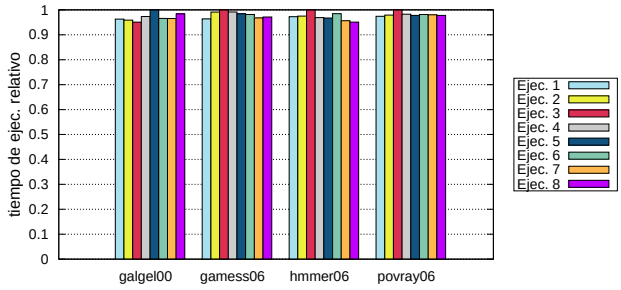

Figure A.2: Normalized completion times of the different applications of the workload permutations running under RR scheduler.

Note that the application's completion times are normalized to the slowest execution of each benchmark across the 8 workload executions. As it can be observed in the figure, an application may suffer a huge variation in its completion time (up to 30%) relative to other executions other `SCHED_HMP`.

It is worth to remark that this variability is not present in scheduling algorithms proposed for asymmetric systems by the scientific community [3, 15, 34, 31]. In particular, the Figure A.2 shows the results of the same workload previously discussed but using a scheduling algorithm that makes a fair distribution of the usage of high throughput cores between applications. This algorithm is known as asymmetry-aware Round-Robin, but for simplicity we will refer to it as RR. The results obtained with this scheduler showcase that the application's completion times between different runs of the same workload are very similar, as it is what the user could expect. Due to the inability of the `SCHED_HMP` scheduler to provide repeatable completion times, the values of different metrics (throughput/justice) are not representative and could lead to wrong conclusions. For that reason, the experimental analysis conducted in this work will not take into account the results of the `SCHED_HMP` scheduler.

A.3. Project goals

Despite the fact that main processor manufacturers are bidding on AMPs, there is not still a reliable support for scheduling in this kind of architectures. The basic goal of this project is to evaluate implementations in Linux of the most relevant scheduling algorithms for AMPs proposed by scientific community. A great deal of this algorithms are already implemented in the *scheduling framework* for Linux, which has been developed by members of the UCM research group ArTeCs. Note that the *scheduling framework* is quite complex (around 25000 lines of code) and until the start of this project was only tested in x86 platforms, like the prototype Intel QuickIA [4]. In this project has been necessary to adapt this framework to a kernel version supported by the asymmetric platform used in our work: the board Odroid XU-4. This development board integrates an ARM big.LITTLE processor. In the chapter 2 can be found a more detailed explanation of this system features.

Energy efficiency a key property of the different algorithms. Sorrowfully, the Odroid XU4 platform has not any register or sensor to obtain power consumption data in runtime. To overcome this limitation, it has been used an external energy-measuring device that acts as a power source at the same time. Despite the existence of an user-level application to obtain consumption data in real time, the device manufacturer does not provide a kernel-level driver to access this information. This support is necessary to obtain fine energy measurements in our experimental study. Therefore, developing a kernel-level driver for this device was one of the duties of this project. The developed driver is currently integrated in the mainline branch of the performance monitoring tool PMCTrack [26], used widely in this work.

A.4. Work plan

As soon as the academic course 2015/2016 begun, we organized a preliminary meeting and defined the goals exposed in the previous section. Furthermore, we defined a work methodology composed by the next principles:

- Establish communication channels (Google Drive, Git and mailing list), meeting habits and division of work.
- Maintain a progressive increment in the load of work and the meeting frequency along the course.
- Keep parallel avenues of work as long as it is possible, informing about results end employed techniques.
- Many important goals were accomplished as a team due to the importance of its acknowledgment by both members.

The planning we set up was constituted by the next tasks:

1. Review documentation about the different tools employed during this work (PMCTrack, PALLOC, scheduling framework,...) and study the necessary concepts to face the development and experimental evaluation (USB drivers, Linux kernel cross compilation,...).
2. Configure the experimental platform to evaluate different scheduling algorithms in the board Odroid XU4
3. Develop a USB driver in Linux to obtain energy consumption measurements using an external device.
4. Conduct an offline analysis to categorize the SPEC CPU applications used in the experiments.
5. Theoretical analysis to define the potential of different scheduling algorithms when optimizing throughput, justice or energy efficiency.
6. Adapt the *scheduling framework* to the kernel version supported by the board Odroid XU4.
7. Evaluate different scheduling algorithms by means of multi-application workloads and discuss the obtained results.
8. Design estimation models based on hardware counters to approximate throughput and energy efficiency factors in the different cores of the asymmetric platform.

These task were not all the time completed in the enunciated order. Some of them were possible to be completed in parallel, such as the third and fourth. Nevertheless, some project phases required teamwork and we were not capable to exploit student-level parallelism efficiently. Due to dependencies between tasks, hardware limitations or the duty to be aware of the applied knowledge.

A.5. Structure of the document

- The **Chapter 2** introduces the set of devices and software tools used to configure the experiment framework and develop the code required in this project.
- The **Chapter 3** describes the set of metrics and scheduling algorithms used as study cases in our project.
- The **Chapter 4** lays out the results from running different benchmarks and workloads using the scheduling algorithms presented in the previous chapter. It is presented a model to estimate different metrics in execution time based in the results obtained.
- The **Chapter 5** exposes the conclusions of this final degree assignment and discusses possible avenues for future work.

Finally, some appendixes are provided. Including: (A) Introduction and (B) Conclusions in English. Lastly, (C) Contributions from each student to this final project .

Apéndice B

Conclusions and future work

B.1. Conclusions

As time goes by mobile devices progressively demand reduced amounts of energy, in order to improve autonomy and performance per watt. Asymmetric multicore processors represent a outstanding alternative to conventional symmetric chips. The AMPs grant the possibility to run multiple types of applications while adapting to their different computing needs.

Most of schedulers on general purpose operating systems are not aware of the underlying asymmetric architectures. A growing number of hardware manufacturers are incorporating this architecture, mainly in their mobile processors such as the big.LITTLE family from ARM. Even though, current operating systems are not fully adapted to this new schemes. Therefore, they do not exploit the features provided by AMPs to their full potential.

In this work, an experimental study has been conducted on the consequences derived from optimizing different types of metrics in the board Odroid XU4, which integrates an ARM big.LITTLE processor. Specifically, it has been paid close attention to the interrelationship between three main properties: throughput, justice and energy efficiency. In order to conduct the analysis, the implementation of several scheduling algorithms in the Linux kernel have been evaluated capable of optimizing one of multiple metrics. Many of these algorithms make scheduling decisions considering the relative benefit in throughput and/or energy efficiency that each application experience from running in a big core versus running in a small one.

Our study reveals that optimizing different metrics, such as justice and global throughput, usually represent antagonistic objectives. Concretely, a substantial degradation of justice has been proven to occur when optimizing global throughput or energy efficiency. In this context, we consider that ACFS and its variants represent an outstanding schedul-

ing strategy. Essentially, most of the evaluated scheduling algorithms offer a fixed trade-off rate between optimizing and degrading the different metrics. Conversely, ACFS is equipped with a `knob` that allows the administrator to configure the degree of justice preserved in the system to the detriment of throughput or energy efficiency.

In the context of optimizing energy efficiency, we evaluated the behaviour of two schedulers. Firstly the EPI-big scheduler, it assigns to high performance cores those applications who show an inferior rate of energy per instruction (EPI). Note that this type of core presents a complex microarchitecture, that generates greater energy consumption than the simpler cores in the platform; therefore reducing the energy consumption of the high performance set of cores is crucial for improving the overall energy efficiency. Secondly, the scheduler known as EEF-Driven maps preferably to high throughput cores those applications who exhibit a greater speedup factor(relative benefit from executing in a big core versus a small one), as well as a lesser energy per instruction rate (EPI) in the high throughput core. Our analysis reveals the following conclusions.

- Optimizing energy delay product (EDP) in asymmetric multicore processors may lead to substantial throughput degradations.
- In most cases, promoting fairness entails the sacrifice of both throughput and energy efficiency.
- Between the evaluated algorithms, the EEF-Driven strategy obtains the best EDP (minimum) for every workload; offering at the same time greater throughput than the EPI-big strategy. These results highlight that substantial reductions in EDP can be obtained by factoring in both applications' speedup factors and big core energy consumption rates.

B.2. Evaluation of the project

The development of this final project has required the utilization of a set of quite complex tools, many of them crafted by the ArTeCS research group with the participation of students from different final projects in previous years. This has been a very interesting final project with a wide theoretical component, that has allowed us to obtain knowledge and aptitudes common in research labours at a professional level. We reckon the results as positive and have reinforced the hypothesis, already verified by other authors [17, 15, 29, 31, 32], that asymmetric systems represent a suitable alternative in the context of high performance computing and mobile devices.

B.3. Future work

The main avenue for future work that could be continued based on the results obtained in this final project would be the implementation of the *dynamic* scheduling algorithms. Our experimental study has only analyzed the *static* versions of this strategies, which do not take into account the different execution phases that the applications go through.

In order to build the *dynamic* algorithms (aware of the different execution phases), it is necessary to provide the scheduler a mechanism to determine in execution time the speedup factors and energy efficiency rates of each thread in the system. This could be achieved implementing the estimation models proposed in the Section 4.3, which are based in hardware counters. These models could be implemented easily by means of a PMCTrack monitoring module [26], which would feed the schedulers with estimations of speedup factors and energy efficiency rates of each thread. Following this approach, the scheduler implementation in the Linux kernel would be fully independent from the underlying architecture. Most importantly, this would enable to keep the base implementation of the *static* scheduling algorithm, but now would receive per-thread estimations of the different parameters needed to make scheduling decisions.

Another main aspect that has not been studied enough in the context of AMPs is the effect of contention in shared resources, such as the cache or the memory bus. In our experimental study we applied cache partitioning to isolate memory intensive applications in a specific region of the cache; in that way, throughput degradation suffered by other cache sensitive applications was mitigated. Despite the fact that many scheduling algorithms that deal with shared-resource contention have been proposed in the context of symmetric multicore systems [41], any of them provides a meticulous analysis of this issue in asymmetric multicore systems. As a consequence, we reckon that combining strategies that prevent shared-resource contention with asymmetry-aware scheduling algorithms could be an interesting research avenue.

An interesting area for future work is the implementation and evaluation of the studied scheduling algorithms in this final project in a real mobile asymmetric platform, based on the Linux kernel (such as the Android operating system). This will allow to evaluate the potential benefits from using asymmetry-aware scheduling algorithms running workloads constituted by different mobile applications. Likewise, specific benchmarks could be crafted to assess the impact of the benefits provided by the different algorithms in battery life. This could suppose an incentive to hardware manufacturers opting for this kind of processors and an impulse to develop a better support for AMP systems in general purpose operating systems.

Apéndice C

Contribuciones de cada participante

En este apéndice cada participante indicará su contribución al proyecto en su respectiva sección. En primer lugar es necesario indicar que muchas de las tareas realizadas en este Trabajo de Fin de Grado han sido completadas en equipo, dada la importancia de que ambos estudiantes conociésemos los procedimientos y herramientas utilizados. De esta forma, las tareas que no se especifica la autoría de un alumno se han hecho en equipo.

C.1. Contribución de Adrián García García

En primer lugar, mi tarea inicial fue el estudio de documentación sobre las herramientas y conceptos en los que se basaría este Trabajo de Fin de Grado. Aunque esta labor de investigación estuvo presente a lo largo de todo el proyecto, puesto que el proyecto ha consistido en trabajo a muy distintos niveles y con una gran variedad de tecnologías. Las principales contribuciones que hice a este proyecto fueron las siguientes:

C.1.0.1. Compilación del kernel

En el proceso de validación de la placa Odroid XU4 como plataforma que soportara nuestro experimental, hubo problemas de compatibilidad entre la versión del kernel en la que se encontraban las herramientas de nuestro entorno experimental (PMCTrack, *framework de planificación*, PALLOC) y el driver CPU-freq (que impedía a los cores funcionar a su máxima frecuencia). Por ello se tuvo que realizar varias compilaciones de diferentes versiones del kernel Linux (3.18 y 3.10) de las que me encargué personalmente. En primer lugar, usamos la versión *3.10.96-pmctrack* para caracterizar las aplicaciones SPEC CPU. Esta versión incluye soporte para la herramienta de gestión de contadores hardware PMCTrack y para realizar medidas de consumo energético usando el software desarrollado en este TFG. Una vez se disponía de la clase de planificación AMP integra-

da, procedí a compilar de nuevo el kernel, resultando la versión final que usamos en este proyecto. La versión *3.10.96-linux-amp*, que incluye el código del Framework de Planificación más el soporte a la herramienta PMCTrack y PALLOC. La problemática de la versión del kernel se explica mas detalladamente en la Sección 2.

C.1.0.2. Migración clase de planificación AMP

Otra de mis aportaciones fue la integración de la clase de planificación AMP en la versión del kernel compatible con la plataforma Odroid XU4. Una vez se encontró una versión del kernel compatible, fue necesario portar el *framework de planificación* que contenía todos los algoritmos que íbamos a usar más adelante en los experimentos con cargas de trabajo multiprogramadas. Con este fin, me encargué de la creación de la clase de planificación AMP. No fue una tarea sencilla, ya que requirió la modificación de muchos archivos y estructuras en diferentes partes de las fuentes del kernel Linux. El procedimiento de crear la clase de planificación se describe más detalladamente en la Sección 3.3.1.

C.1.0.3. Driver standalone

Me encargué de implementar la versión inicial del driver USB *standalone* del dispositivo Odroid Smart Power. Aunque hicimos un trabajo conjunto para decidir la forma de implementarlo, inicialmente no conseguimos una versión que compilase así que decidimos continuar por nuestra cuenta para construir un esqueleto básico funcional. Finalmente, conseguí desarrollar una versión inicial que compilase y más tarde continuamos trabajando en equipo para mejorarla y poder cumplir con la funcionalidad requerida para obtener mediciones de energía. Se puede encontrar más información sobre el driver *standalone* en la Sección 2.3.1.1.

C.1.0.4. Generación de MRCs

También me encargué de la generación de MRCs con `matplotlib`. Una vez instalada la versión de PALLOC en el kernel de la Odroid XU4, procedí a estudiar la documentación sobre PALLOC [40]. Una vez disponía de ciertos conocimientos, experimenté con la asignación de diferentes particiones de cache a aplicaciones en nuestra plataforma experimental, y creé un tutorial para configurar el lanzamiento de aplicaciones con diferentes tamaños de particionado. Tras esto, conté con la ayuda de mi compañero que creó una serie de scripts para sistematizar la obtención de información de varias ejecuciones que variaban el tamaño de cache disponible para una misma aplicación. Una vez teníamos la información, usé la librería de *python* `matplotlib` para crear unas gráficas de MRC (*Miss-Rate Curve*) que permitiesen ilustrar los diferentes grados de sensibilidad que muestran las aplicaciones. Las propias gráficas y su discusión se pueden encontrar en la Sección 2.2.2.

C.1.0.5. Scripts de resultados de conjuntos de eventos

Con el objetivo de plantear los modelos de estimación con WEKA, fue necesario ejecutar los SPEC CPU 2000 y 2006 midiendo varios conjuntos de eventos para las mismas ventanas de instrucciones (500 millones de tamaño de ventana de instrucciones). Concretamente, se necesitaban 2 conjuntos de eventos para el cluster big y 3 para el cluster small (debido a que estos últimos tipos de core disponen de menor cantidad de contadores hardware). Una vez se obtuvieron los resultados de cada core, preparé 5 scripts para interpretar los resultados obtenidos teniendo en cuenta las fórmulas para aislar el consumo neto de cada aplicación en cada tipo de core. Son scripts en *python* que definen un conjunto de funciones que obtienen el valor de cada contador hardware de los logs de Het-Harness y operan con ellos para obtener métricas de alto nivel. Para más información consultar la Sección 4.3.

C.1.0.6. Traducciones a inglés

Me ocupe en solitario de las traducciones al inglés, debido principalmente a que me encuentro a punto de presentarme al examen de Advanced de Cambridge (Nivel C1) y me sirve de práctica al mismo tiempo que cumplo un requisito del Trabajo de Fin de Grado. En concreto, me encargué de traducir al Inglés los capítulos de Introducción y Conclusiones (se pueden encontrar en los apéndices A y B). También realicé el estudio y recopilación de información procedente de varios *papers* de diferentes revistas y congresos [31, 32, 26, 40]. Gracias a ello, obtuvimos unos conocimientos críticos para nuestro Trabajo de Fin de Grado sobre la planificación en sistemas asimétricos y diferentes técnicas que permitían mitigar los efectos de la contención de recursos compartidos usando la herramienta PALLOC.

C.2. Contribución de Álvaro Sanz del Río

La principal tarea de este proyecto ha sido el estudio de cada uno de los problemas o retos que se nos han ido planteando en su transcurso como la problemática con el entorno de trabajo, la complejidad de las herramientas aplicadas en el análisis experimental, etc. Las principales contribuciones que hice en este trabajo han sido las siguientes:

C.2.0.7. Procesamiento de resultados

Una vez definidas las métricas para el análisis experimental y el entorno estaba capacitado para la ejecución de aplicaciones individuales o cargas de trabajo multiprogramadas comenzó la realización de experimentos. Mi cometido en esta parte del proyecto era el de ejecutar hilos de aplicaciones individuales o ejecuciones sistemáticas de cargas multiprogramadas con las herramientas descritas en el Capítulo 2.2 para después realizar los análisis concluyentes sobre las caracterizaciones de las distintas aplicaciones y los comportamientos de los distintos algoritmos de planificación explicados en la Sección 3.4. Durante el transcurso del proyecto, una vez que se terminaba una ejecución en un

experimento se generaban en unos *logs* con los resultados, estos datos se procesaban y se generaban gráficas para su posterior análisis. Estos experimentos producían grandes cantidades de datos, por lo que se han tenido que utilizar herramientas de shell scripting (programación de scripts en bash) o scripts en *python* para poder facilitar este trabajo. Una vez extraídos los datos, el cometido era generar gráficas para facilitar el análisis. Como visualización preliminar de los análisis, muchas de las gráficas se presentaban en *excel* de forma rápida para poder discutir si el experimento necesitaba más de una prueba, por diferentes motivos (p.ej. Un pico de energía debido al mal uso del ventilador, fallo en el uso de alguna herramienta, desactivación errónea de cores, etc).

C.2.0.8. Primeros experimentos con Het-Harness

Mi contribución en esta parte del proyecto fue la del estudio y puesta en práctica de la herramienta Het-Harness. En primer lugar estudié toda la documentación proporcionada para entender su funcionamiento y el uso que teníamos que hacer de la herramienta a lo largo de nuestro TFG. Después hice varias pruebas de ejecución con aplicaciones individuales y cargas de trabajo para poner en práctica lo aprendido en la documentación y para terminar redacté un pequeño tutorial que nos serviría de ayuda en el lanzamiento de experimentos el resto del proyecto. En este tutorial se describían los scripts que necesitábamos en los experimentos y el objetivo de cada uno de ellos. El darnos soporte para el lanzamiento de *benchmarks* ampliamente usados como SPEC CPU2006 y CPU2000, proporcionarnos scripts de procesamiento de los *logs* generados por los lanzadores de cargas de trabajo para obtener métricas de alto nivel (justicia, productividad, consumo energético, etc) y la integración con PMCTrack para la monitorización de eventos hardware en la ejecución de distintos *benchmarks* nos facilitó toda la parte de análisis experimental descrito en el Capítulo 4.

C.2.0.9. Uso de la herramienta PALLOC

Con la herramienta PALLOC[40] (explicada en la Sección 2.2.2) se consiguió mitigar los problemas de contención de memoria cache en las ejecuciones de las cargas de trabajo. Después de un conjunto de pruebas y estudio de la herramienta. Mi contribución en este punto, partiendo de unos conocimientos previos y de un tutorial proporcionado por mi compañero, consistió en la elaboración de un script en bash donde se medía mediante la herramienta PMCTrack los fallos de cache por cada mil instrucciones y las instrucciones retiradas. Este script hacía uso de esta herramienta por cada una de las particiones realizadas en la cache, en concreto se realizaron 16 *bins* (particiones) de 128 KB cada uno. Con los resultados obtenidos Adrián generó las gráficas MRC (*Memory Restricted Cache*) para analizar los distintos tipos de comportamiento de las aplicaciones según su sensibilidad al uso en mayor o menos medida de la memoria.

C.2.0.10. Uso de la herramienta Weka

Weka es una plataforma de software para el aprendizaje automático y la minería de datos escrito en Java, el paquete contiene una colección de herramientas de visualización y algoritmos para análisis de datos y modelado predictivo, unidos a una interfaz gráfica de usuario para acceder fácilmente a sus funcionalidades. Esta herramienta también está dotada de funcionalidad por medio de línea de comandos. En nuestro proyecto utilicé está herramienta tanto la interfaz gráfica como desde terminal para generar los modelos de estimación descritos en la Sección 4.3. En un principio utilicé la interfaz gráfica para familiarizarme con el entorno y observar las estimaciones que obtenía, buscando unos coeficientes de correlación lo más elevados posible. Para generar los modelos de estimación y con los resultados hacer gráficas, tuve que adaptar un script proporcionado por Juan Carlos debido a que su versión de WEKA era anterior a la descargada de la página oficial. Con este cambio, el script era capaz de, con las estimaciones proporcionadas por la herramienta, generar las gráficas de estimación de los factores EEF y SF para los cores big y small.

C.2.0.11. Generación de gráficas mediante Matplotlib y análisis con porcentajes de mejora

Cuando se obtuvieron los resultados de los distintos análisis realizados en todo el proyecto, se generaron las gráficas que se ven a lo largo de la memoria con Gnuplot o librerías de *python* como `matplotlib`. En este punto realicé un estudio intensivo de los resultados de las cargas de trabajo multiprogramadas sacando porcentajes de mejoras de los distintos algoritmos de planificación y cada una de sus cargas para poder compararlos de una manera más precisa. En el segundo estudio con las cargas de trabajo a las que se les aplicó el barrido de los dos factores del planificador ACFS (*Unfairness Factor* y *EDP Factor*) para poder comparar los resultados de forma fácil, lo mejor era que la injusticia y la otra propiedad con la que se quería comparar (rendimiento o la eficiencia energética) estuvieran en la misma gráfica. Para esto hice uso de la librería `matplotlib` con la que generé las gráficas con las tres cargas de trabajo donde se ve la degradación de la justicia producido por la mejora de el rendimiento o la eficiencia energética.

Bibliografía

[1] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's Law through EPI Throttling. In *Proceedings of ISCA 05*, pages 298–309, Wisconsin, USA, 4-8 June 2005. IEEE Computer Society, Washington, DC, USA.

[2] ARM. Benefits of the big.LITTLE Architecture. `http://www.arm.com/files/downloads/Benefits_of_the_big.LITTLE_architecture.pdf`. Accessed: 2015-01-10.

[3] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *Proceedings of CF 06*, pages 29–40, Ischia, Italy, 2-5 May 2006. ACM, New York.

[4] N. Chitlur et al. QuickIA: Exploring heterogeneous architectures on real prototypes. In *Proceedings of HPCA 12*, pages 1–8, New Orleans, LA, 25-29 February 2012. IEEE Computer Society, Washington, DC, USA.

[5] W. Cohen. Tuning programs with oprofile. *Wide Open Magazine*, 1:53–62, 2004.

[6] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of ASPLOS 10*, pages 335–346, Pittsburgh, PA, 13-17 March 2010. ACM, New York.

[7] R. Gabor, S. Weiss, and A. Mendelson. Fairness and throughput in switch on event multithreading. In *Proceedings of MICRO 06*, pages 149–160, Orlando, FL, 9-13 December 2006. IEEE Computer Society Washington, DC, USA.

[8] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, Sep 1996.

[9] M. Gschwind. The Cell Broadband Engine: exploiting multiple levels of parallelism in a chip multiprocessor. *Int. J. Parallel Program.*, 35(3):233–262, 2007.

[10] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *SIGKDD Explor. Newsl.*, 11:10–18, 2009.

[11] Hardkernel. Odroid xu4: User manual. `http://magazine.odroid.com/odroid-xu4/`. Accessed: 2016-05-16.

[12] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, 2008.

[13] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Proceedings of IEEE Symposium on Low Power Electronics*, pages 8–11, San Diego, CA, 10-12 Oct 1994.

[14] S. Jarp, R. Jurga, and A. Nowak. Perfmon2: a leap forward in performance monitoring. *Journal of Physics: Conference Series*, 119:042017, 2008.

[15] D. Koufaty, D. Reddy, and S. Hahn. Bias Scheduling in Heterogeneous Multi-core Architectures. In *Proceedings of Eurosys 10*, pages 125–138, Paris, France, 13-16 April 2010. ACM, New York.

[16] R. Kumar et al. Single-ISA Heterogeneous Multi-Core Architectures: the Potential for Processor Power Reduction. In *Proc. of MICRO 36*, 2003.

[17] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proceedings of ISCA 04*, pages 64–75, Munich, Germany, 19-23 June 2004. IEEE Computer Society, Washington, DC, USA.

[18] T. Li, P. Brett, R. Knauerhase, and D. Koufaty. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *Proceedings of HPCA 10*, pages 1–12, Bangalore, India, 9-14 January 2010. IEEE Computer Society Press, Los Alamitos, CA.

[19] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of MICRO '07*, pages 146–160, Chicago, IL, 1-5 December 2007. IEEE Computer Society Washington, DC, USA.

[20] Perf. Perf wiki tutorial on perf. `https://perf.wiki.kernel.org/index.php`, 2015. Accessed: 2015-01-20.

[21] PMCTrack. project official website. `http://pmctrack.dacya.ucm.es/`.

[22] PMCTrack. Source code repository at Github. `https://github.com/jcsaezal/pmctrack`, 2015.

[23] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of MICRO 06*, pages 423–432, Orlando, FL, 9-13 December 2006. IEEE Computer Society Washington, DC, USA.

[24] M. Rasmussen. Task placement for heterogeneous mp systems. `https://lwn.net/Articles/517250/`, 2012.

[25] D. Reddy, D. Koufaty, P. Brett, and S. Hahn. Bridging functional heterogeneity in multicore architectures. *SIGOPS Oper. Syst. Rev.*, 45(1):21–33, Feb. 2011.

[26] J. C. Saez, J. Casas, A. Serrano, R. Rodríguez-Rodríguez, F. Castro, D. Chaver, and M. Prieto-Matias. An OS-oriented performance monitoring tool for multicore systems. In *Proc. of Euro-Par 2015: Parallel Processing Workshops*, pages 697–709, Cham, 2015. Springer International Publishing.

[27] J. C. Saez et al. Delivering fairness and priority enforcement on asymmetric multicore systems via os scheduling. In *Proc. of the ACM SIGMETRICS '13*, pages 343–344, 2013.

[28] J. C. Saez et al. Exploring the throughput-fairness trade-off on asymmetric multicore systems. In *To appear in 2nd Workshop on Runtime and Operating Systems for the Many-core Era (ROME), held in conjunction with Euro-Par 2014*, 2014.

[29] J. C. Saez, A. Fedorova, D. Koufaty, and M. Prieto. Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems. *ACM TOCS*, 30(2):6:1–6:38, Apr. 2012.

[30] J. C. Saez, A. Pousa, F. Castro, D. Chaver, and M. Prieto-Matias. Exploring the throughput-fairness trade-off on asymmetric multicore systems. In *Proceedings of Euro-Par 14: Parallel Processing Workshops*, pages 326–337, Porto, Portugal, 25-26 August 2014. Springer-Verlag, Berlin.

[31] J. C. Saez, A. Pousa, F. Castro, D. Chaver, and M. Prieto-Matias. ACFS: A completely fair scheduler for asymmetric single-ISA multicore systems. In *Proceedings of the ACM SAC 15*, pages 2027–2032, Salamanca, Spain, 13-17 April 2015. ACM, New York.

[32] J. C. Saez, A. Pousa, A. E. De Giusti, and M. Prieto-Matias. On the interplay between throughput, fairness and energy efficiency on asymmetric multicore processors. *Submitted to an International Journal for review*, 2016.

[33] J. C. Saez, A. Pousa, R. Rodriguez, F. Castro, and M. Prieto-Matias. Delivering performance counter monitoring support to the OS scheduler. *Computer Journal (accepted with minor revisions)*, 2016.

[34] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A Comprehensive Scheduler for Asymmetric Multicore Systems. In *Proceedings of Eurosys 10*, pages 139–152, Paris, France, 13-16 April 2010. ACM, New York.

[35] J. C. Saez, D. Shelepov, A. Fedorova, and M. Prieto. Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems. *J. Parallel Distrib. Comput.*, 71:114–131, January 2011.

[36] D. Shelepov, J. C. Saez, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. HASS: a Scheduler for Heterogeneous Multicore Systems. *ACM Operating Systems Review*, 43(2):66–75, 2009.

[37] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. In *Proceedings of ASPLOS 09*, pages 121–132, Washington, DC, 7-11 March 2009. ACM, New York.

[38] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout. Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In *Procedings of PACT 13*, pages 177–187, Edinburgh, Scotland, 7-11 September 2013. ACM, New York.

[39] V. Weaver. Linux perfevents features and overhead. In *Proceedings of International Workshop on Performance Analysis of Workload Optimized Systems*, pages 80–80, Austin, TX, 21 April 2013. IEEE Computer Society Press, Los Alamitos, CA.

[40] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 155–166. IEEE, 2014.

[41] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.*, 45(1):4:1–4:28, Dec. 2012.